Security Supplement
to the
Software Communications Architecture Specification

Attachment 1
Security Application Program Interface
Service Definition

Revision Summary

1.0	Initial Release

Table of Contents

List of Figures

List of Tables

1 INTRODUCTION.

1.1 OVERVIEW.

This document specifies the application program interfaces (APIs) for security services that are required in a secure JTRS-compliant radio.

1.2 MODES OF SERVICE.

The JTRS security service does not identify global modes but does identify modes within certain services. The following paragraphs enumerate the modes defined this document.

1.2.1 Fill Modes.

The security API defines four modes for filling a radio. Three of the modes are entered by configuring the fill port and are mutually exclusive for that port. They are DS-101, DS-102 and RS-232. The fourth mode fills the radio from a file that does not enter the system through the fill port.

1.2.1.1 DS-101 Fill Mode.

The DS-101 fill mode supports the DS-101 fill protocol at the fill port. This mode is essentially autonomous once the information load has commenced. The fill information may contain multiple keys, algorithms and TRANSEC information.

1.2.1.2 DS-102 Fill Mode.

The DS-102 fill mode supports the DS-102 fill protocol at the fill port. This mode requires human intervention and the API is defined to reflect this.

1.2.1.3 RS-232 Fill Mode.

The DS-102 fill mode supports the DS-102 fill protocol at the fill port. This mode is similar to the Bus Fill mode. A file is transferred through the fill port.

1.2.1.4 Bus Fill Mode.

The Bus fill mode supports input of fill information from a file which enters the system like other software. The file may contain keys, TRANSEC and other information in an encrypted file. This fill is passed to the cryptographic module using the Bus service.

1.2.2 Crypto Channel Modes.

When a crypto channel is created it is created to operate in one of five modes. The five modes are defined in the following paragraphs.

1.2.2.1 Simplex Receive Mode.

The channel is configured for received only. The crypto does not allocate any resources to support transmit.

1.2.2.2 Half-Duplex Mode.

The channel is configured for transmit and receive. The crypto allocates its resources to support both transmit and receive, but not simultaneously.

1.2.2.3 Full-Duplex Mode.

The channel is configured for transmit and receive. The crypto allocates its resources to support both transmit and receive simultaneously.

1.2.2.4 Red Side Mode.

The channel is configured for red side only behavior. The crypto allocates its resources such that the results of encryption or decryption of data entering the red side exit on the red side.

1.2.2.5 Black Side Mode.

The channel is configured for black side only behavior. The crypto allocates its resources such that the results of encryption or decryption of data entering the black side exit on the black side.

1.3 SERVICE STATES.

States are described in section 5.

1.4 REFERENCED DOCUMENTS.

None.

2 UUID.

To be assigned upon formal release of this document.

3 SERVICES.

The entirety of the JTRS Security Service can logically be represented as composed of service groups. The Unified Modeling Language (UML) package diagram in Figure 3-1 depicts the JTRS Security Service and its service groups as packages. Each of these groups represents a functional area of security that directly or indirectly supports secure JTRS radio operation. Each functional group contains one or more related services. The service groups also provide naming scope for services within different groups that are related. An example of this is a management service. Several of the service groups contain a management service. The general behavior of this service is the same across certain groups. What differentiates the specific behavior is the type of element being managed, which is identified by the service group (e.g. Key).

Figure 3-1. JTRS Security Service Groups

The individual primitives that may flow between the Service User and Service Provider define each service within a service group. The services and primitives are tabulated in Table 3-1 and described more fully in the remainder of this section.

Table 3-1. Cross-Reference of Services and Primitives.

Service Group	Service	Primitives
Security	Management	ZEROIZE_ALL
Fill	Port	FILL_PORT_CONFIGURE, FILL_PORT_ENABLE, FILL_PORT_DISABLE, FILL_PORT_LOAD
	Port User	FILL_ PORT_SIGNAL_ASSIGN_ID, FILL_ PORT_SIGNAL_LOAD, FILL_ PORT_SIGNAL_CONNECT,
	Bus	FILL_BUS_LOAD
	Management	FILL_ZEROIZE, FILL_ZEROIZE_ALL, FILL_GET_IDS, FILL_EXPIRY
Algorithm	Management	ALG_ZEROIZE, ALG_ZEROIZE_ALL, ALG_GET_IDS, ALG_EXPIRY
Certificate	Management	CERT_ZEROIZE, CERT_ZEROIZE_ALL, CERT_GET_IDS, CERT_EXPIRY
Crypto	Control	CRYPT_CREATE_CHAN, CRYPT_DESTROY_CHAN, CRYPT_GET_CHAN_CONFIG, CRYPT_START_CHAN, CRYPT_STOP_CHAN, CRYPT_RESET_CHAN, CRYPT_RESET
	Encrypt/Decrypt	CRYPT_ENCRYPT, CRYPT_DECRYPT, CRYPT_ENCRYPT_WITH_ ID, CRYPT_DECRYPT_WITH_ ID, CRYPT_TRANSFORM_REQ, CRYPT_TRANSFORM_REQ_WITH_ID
Key	Management	KEY_ZEROIZE, KEY_ZEROIZE_ALL, KEY_GET_IDS, KEY_EXPIRY, KEY_UPDATE, KEY_GET_UPDATE_COUNT, KEY_STORE_KEY

Service Group	Service	Primitives
TRANSEC	Control	TRAN_CREATE_CHAN, TRAN_GET_CHAN_CONFIG, TRAN_DESTROY_CHAN
	Key Stream	TRAN_GEN_KEY_STREAM, TRAN_GEN_NEXT_KEY_STREAM
	Management	TRAN_ZEROIZE, TRAN_ZEROIZE_ALL, TRAN_GET_IDS, TRAN_EXPIRY, TRAN_STORE , TRAN_GET_FILL
Policy	Management	POL_ZEROIZE, POL_ZEROIZE_ALL, POL_GET_IDS, POL_EXPIRY, POL_GET_POLICY
Integrity and Authentication	Control	IA_CREATE_CONTEXT, IA_DESTROY_CONTEXT
	Digital Signatures	IA_SIGN_FILE, IA_VERIFY_FILE, IA_HASH, IA_SIGN_HASH, IA_VERIFY_HASH
Alarm	User	ALARM_SIGNAL
Time	Management	TIME_SET_TOD, TIME_GET_TOD, TIME_SET_DATE, TIME_GET_DATE
GPS	Management	GPS_ZEROIZE, GPS_ZEROIZE_ALL, GPS_GET_IDS, GPS_EXPIRY, GPS_STORE , GPS_GET_FILL

3.1 GAINING ACCESS TO SECURITY SERVICES.

Figure 3-2 shows an SCA component which is a CF::*Device*. The device is a logical representation of a cryptographic subsystem. The device has several ports. Each port represents a security service. For example the Key Management Service is at one port while the Crypto Control Service is at another. Each of these ports has an identifier. When a Security Service User needs to gain access to a service it invokes the *getPort* operation on the security device with the port identifier as input. The *getPort* operation returns the object reference of the service provider which can then be passed to the service user through the CF::*Port*::*connectPort* operation. The service user can then invoke the primitives that comprise the service.

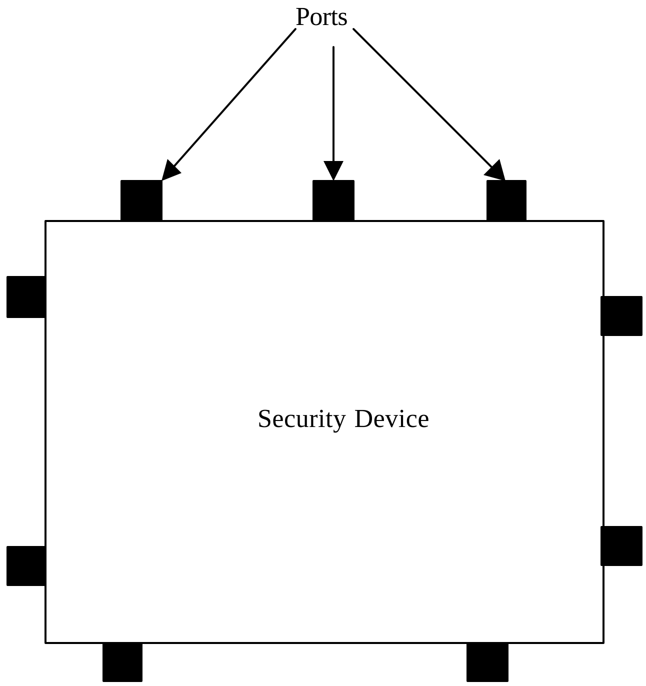

Figure 3-2. JTRS Security Device

3.2 SECURITY.

Security at the top level has one service, a management service.

3.2.1 Management.

Figure 3-3 illustrates a Service User invoking the ZEROIZE_ALL primitive of the Security Management Service. The ZEROIZE_ALL primitive zeroizes all elements of fill information. It is equivalent to invoking the individual zeroize all primitives of the Algorithm, Certificate, Key, Policy and TRANSEC Management Services.

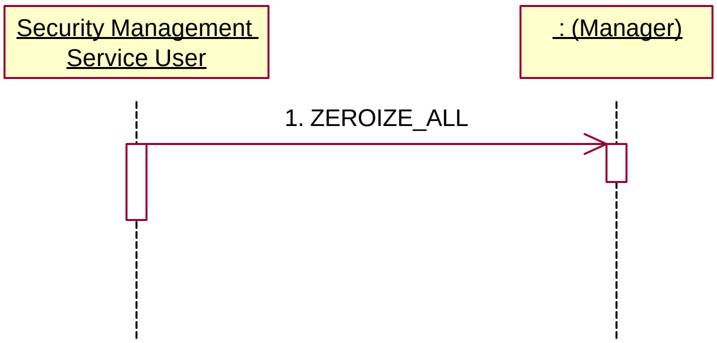

Figure 3-3. Sequence Diagram: Zeroizing all Elements within a Security Service

3.3 FILL.

The fill services defined in the security API consist of services to get fill information into a JTR and to manage that information.

3.3.1 Port and Port User Services.

Figure 3-4 shows a sequence diagram of a DS-101 or RS-232 type fill using the Port and Port User services. The Port service is implemented by the security service. The Port User service is implemented by the user of the security service (e.g. the human machine interface (HMI) software).

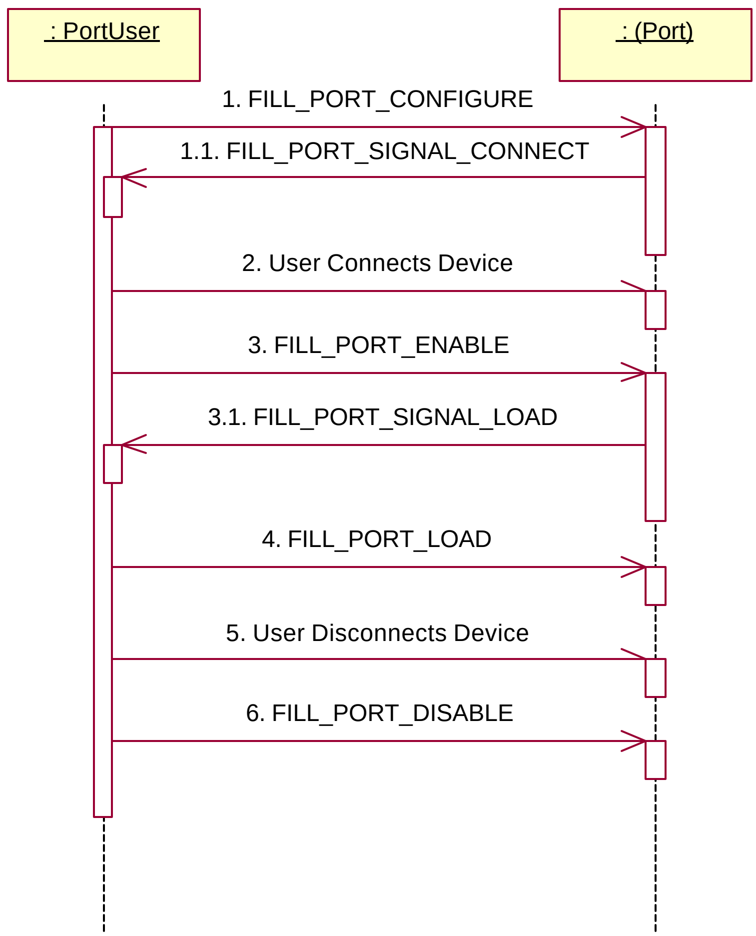

Figure 3-4. Sequence Diagram: DS-101 or RS-232 Fill using Port and Port User Services

1. The Port User invokes the FILL_PORT_CONFIGURE primitive that configures the Fill Port for a mode of operation which in this case is DS-101 or RS-232.

1.1 The Fill Port invokes the FILL_PORT_SIGNAL_CONNECT primitive on the Port User to notify the user to connect the fill device.

2. The user connects the fill device to the Fill Port.
3. The Port User enables the Fill Port by invoking the FILL_PORT_ENABLE primitive on the Port.

3.1 The Fill Port invokes the FILL_PORT_SIGNAL_LOAD primitive on the Fill Port User to notify the user to begin the loading from the fill device.

4. The Fill Port User invokes the FILL_PORT_LOAD primitive to start the load through the fill port.
5. The user disconnects the fill device from the Fill Port.
6. The Port User invokes the FILL_PORT_DISABLE primitive on the Fill Port.

Figure 3-5 shows a sequence diagram of a DS-102 fill using the Port and Port User services. The Key Management Service is included for clarity. The TRANSEC Management service has an equivalent primitive. A DS-102 fill requires more human intervention than either a DS-101 or RS-232 type fill. The Fill Port User service includes the additional primitives to support this type of fill.

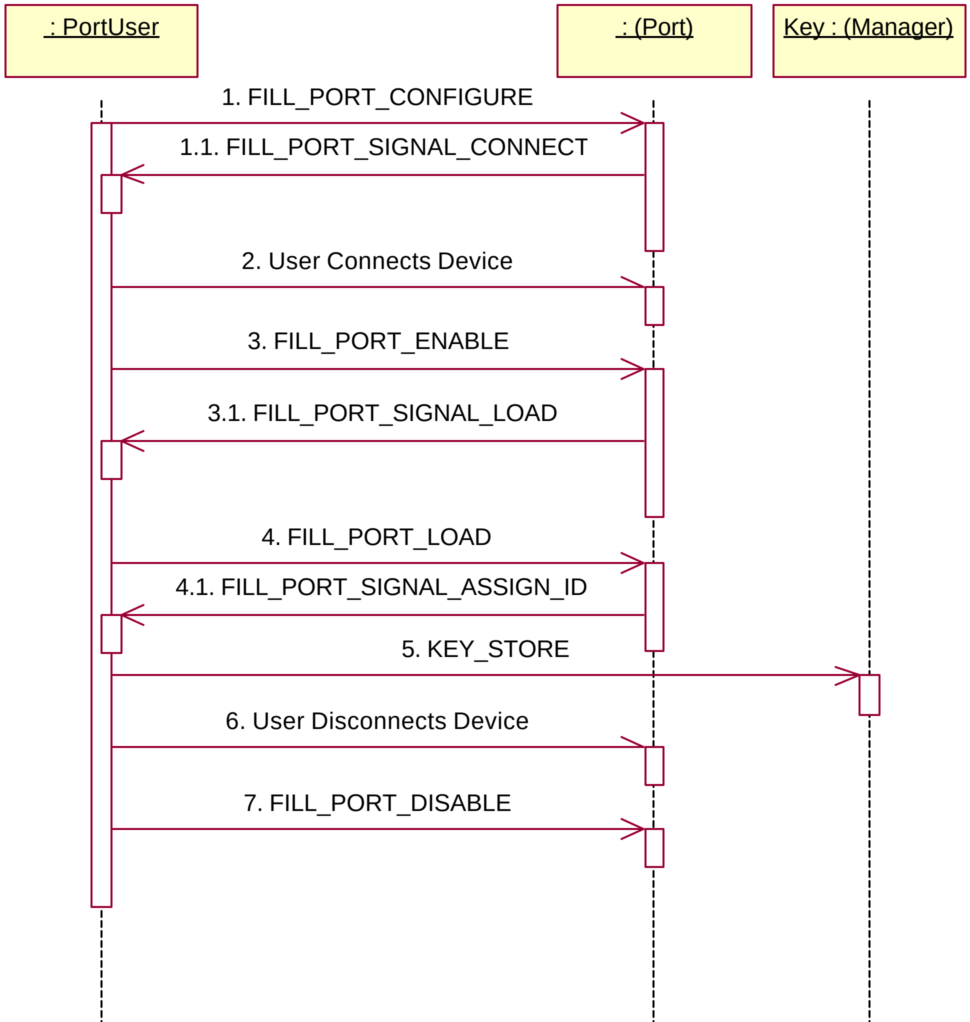

Figure 3-5. Sequence Diagram: DS-102 Fill using Port and Port User Services

1. The Port User invokes the FILL_PORT_CONFIGURE primitive which configures the Fill Port for DS-102.

1.1 The Fill Port invokes the FILL_PORT_SIGNAL_CONNECT primitive on the Port User to notify the user to connect the fill device.

2. The user connects the fill device to the Fill Port.

3. The Port User enables the Fill Port by invoking the FILL_PORT_ENABLE primitive on the Port.

3.1 The Fill Port invokes the FILL_PORT_SIGNAL_LOAD primitive on the Fill Port User to notify the user to begin the loading from the fill device.

4. The Fill Port User invokes the FILL_PORT_LOAD primitive to start the load through the fill port.

4.1 The Fill Port invokes the FILL_PORT_ASSIGN_ID primitive on the Fill Port User to notify him to assign an ID to the fill element, in this case a key.

5. The Fill Port User invokes the STORE_KEY primitive on the Key Management Service with the name ID the user has assigned.

6. The user disconnects the fill device from the Fill Port.

7. The Port User invokes the FILL_PORT_DISABLE primitive on the Fill Port.

3.3.2 Bus Service.

The Bus Service allows the Service User to fill the radio from a file resident on an SCA compliant file system. Figure 3-6 illustrates the user of the bus service invoking the FILL_BUS_LOAD primitive to accomplish the fill. The file name and its location are input as part of the primitive.

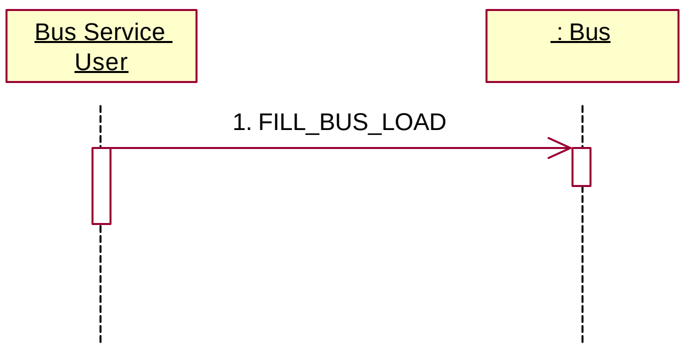

Figure 3-6. Sequence Diagram: Filling the Radio from a File using the Bus Service

3.3.3 Management Service.

The Fill Management Service is not implemented directly. This service provides a set of primitives that are common across a set of management services. The Fill Management Service is inherited, specialized and extended by other services. It is in these other services where the implementation will reside.

Figure 3-7 illustrates a Service User invoking the FILL_ZEROIZE primitive of the Fill Management Service. The FILL_ZEROIZE primitive zeroizes a single element of fill information. The type of fill information depends on the inheriting service.

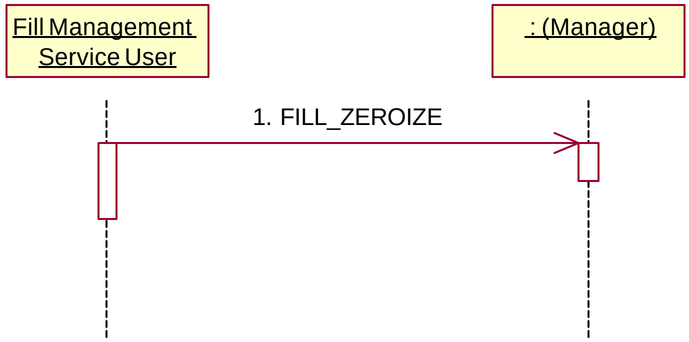

Figure 3-7. Sequence Diagram: Zeroizing an Element using the Management Service

Figure 3-8 illustrates a Service User invoking the FILL_ZEROIZE_ALL primitive of the Fill Management Service. The FILL_ZEROIZE_ALL primitive zeroizes all elements of fill information with in a service. The type of fill information depends on the inheriting service.

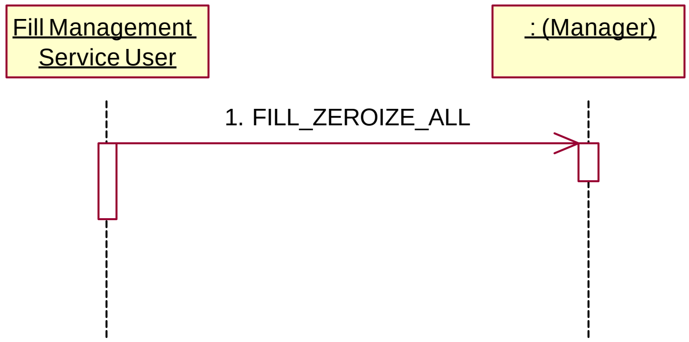

Figure 3-8. Sequence Diagram: Zeroizing all Elements using the Management Service

Figure 3-9 illustrates a Service User invoking the FILL_GET_IDS primitive of the Fill Management Service. The FILL_GET_IDS primitive gets the identifiers of all the elements of fill information with in a service. The type of fill information depends on the inheriting service.

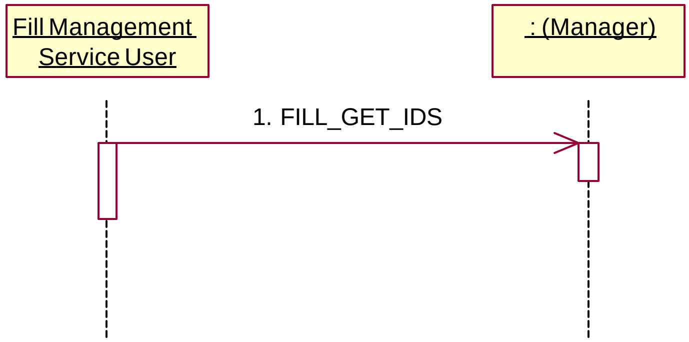

Figure 3-9. Sequence Diagram: Getting the Identifiers of all Elements using the Management Service

Figure 3-10 illustrates a Service User invoking the FILL_EXPIRY primitive of the Fill Management Service. The FILL_EXPIRY primitive gets the date and time of expiration for a single element and returns them to the Service User.

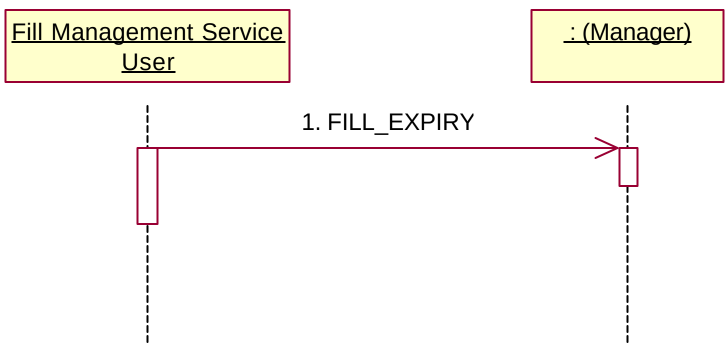

Figure 3-10. Sequence Diagram: Getting Expiration Info using the Fill Management Service

3.4 ALGORITHM.

Algorithms encompass both encryption and classified TRANSEC algorithms. Algorithms require only one service, a management service. The Crypto Control Service and TRANSEC Control Service instantiate traffic and key stream generation channels with the algorithms managed by the algorithm management service. The logical separation of the service that manages algorithms and the services that use algorithms imposes no such separation in the implementation.

3.4.1 Management Service

The Algorithm Management Service is a specialization of the Fill Management Service with no additional primitives. The ALG_ZEROIZE, ALG_ZEROIZE_ALL, ALG_GET_IDS and ALG_EXPIRY primitives have the same behavior as the corresponding FILL_ZEROIZE, FILL_ZEROIZE_ALL, FILL_GET_IDS and FILL_EXPIRY primitives where the elements are algorithms.

3.5 CERTIFICATE.

The Integrity and Authentication Service uses certificates for generating and verifying Digital Signatures. In addition they may be used for key exchanges such as Firefly. Certificates require only one service, a management service.

3.5.1 Management Service.

The Certificate Management Service is a specialization of the Fill Management Service with one additional primitive. The CERT_ZEROIZE, CERT_ZEROIZE_ALL, CERT_GET_IDS and CERT_EXPIRY primitives have the same behavior as the corresponding FILL_ZEROIZE, FILL_ZEROIZE_ALL, FILL_GET_IDS and FILL_EXPIRY primitives where the elements are certificates.

3.6 CRYPTO.

Cryptographic (COMSEC) functionality is encompassed in the Crypto Control and Encrypt/Decrypt services.

3.6.1 Control Service.

The Crypto Control Service covers channel creation, destruction, starting, stopping, resetting and registration for crypto alarm notification.

Figure 3-11 illustrates a Service User invoking the CRYPT_CREATE_CHAN primitive of the Crypto Control Service. The CRYPT_CREATE_CHAN causes the Crypto Control Service to allocate internal resources and create a crypto channel. The channel type, algorithm, key(s), mode(s) and properties (e.g. straps) are specified as part of the primitive. In addition a certificate may be specified for establishment of a security association such as in a Firefly exchange. The CRYPT_CREATE_CHAN primitive returns an opaque channel identifier to the Service User.

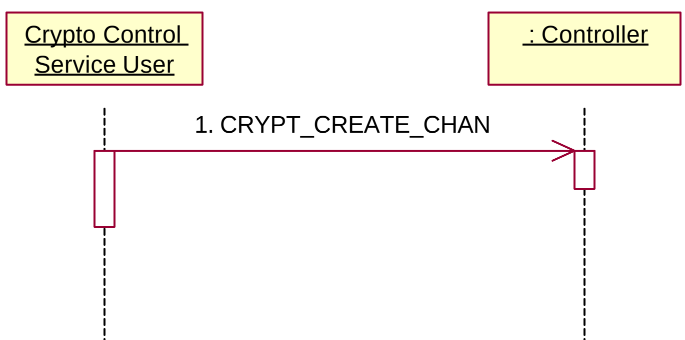

Figure 3-11. Sequence Diagram: Creating a Channel using the Crypto Control Service

Figure 3-12 illustrates a Service User invoking the CRYPT_DESTROY_CHAN primitive of the Crypto Control Service. The CRYPT_DESTROY_CHAN primitive destroys a channel created by the CRYPT_CREATE_CHAN primitive and all cryptographic resources allocated to the channel are released.

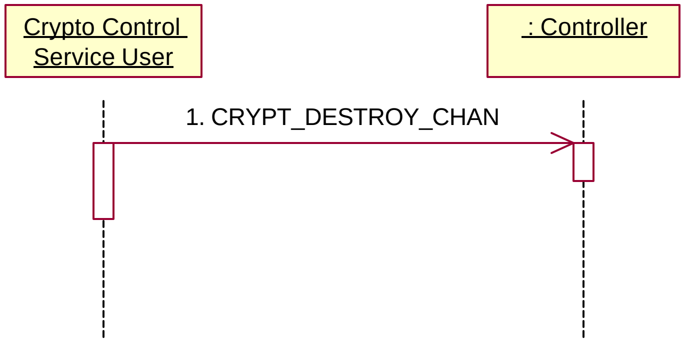

Figure 3-12. Sequence Diagram: Destroying a Channel using the Crypto Control Service

Figure 3-13 illustrates a Service User invoking the CRYPT_GET_CHAN_CONFIG primitive of the Crypto Control Service. The CRYPT_GET_CHAN_CONFIG primitive returns the configuration information used to create a channel with the CRYPT_GET_CHAN_CONFIG primitive.

Figure 3-13. Sequence Diagram: Getting the Configuration of a Crypto Channel using the Crypto Control Service

Figure 3-14 illustrates a Service User invoking the CRYPT_START_CHAN primitive of the Crypto Control Service. The CRYPT_ START_CHAN primitive is used to start a crypto channel or an individual mode of a crypto channel such as a Firefly exchange.

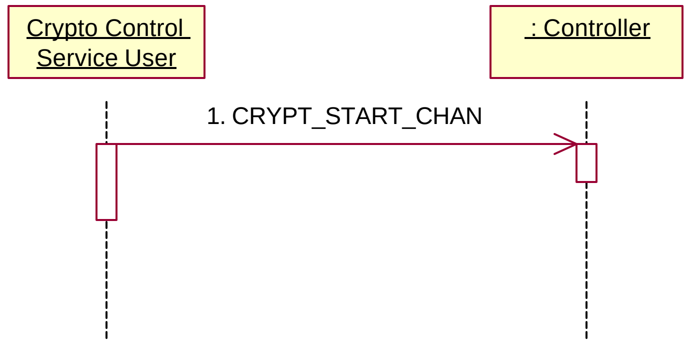

Figure 3-14. Sequence Diagram: Starting a Crypto Channel using the Crypto Control Service.

Figure 3-15 illustrates a Service User invoking the CRYPT_STOP_CHAN primitive of the Crypto Control Service. The CRYPT_ STOP_CHAN primitive is used to stop a crypto channel or an individual mode of a crypto channel such as a Firefly exchange.

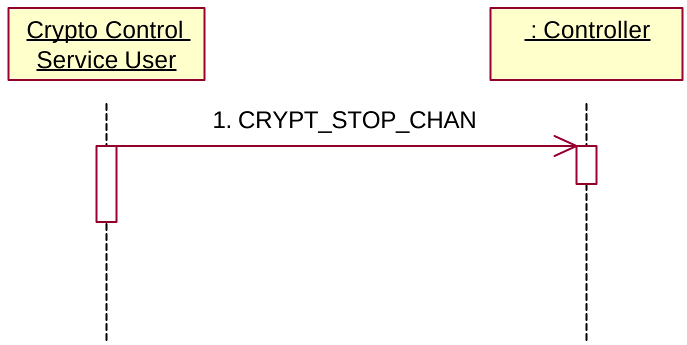

Figure 3-15. Sequence Diagram: Stopping a Crypto Channel using the Crypto Control Service

Figure 3-16 illustrates a Service User invoking the CRYPT_RESET_CHAN primitive of the Crypto Control Service. The CRYPT_ RESET _CHAN primitive is used to reset a crypto channel.

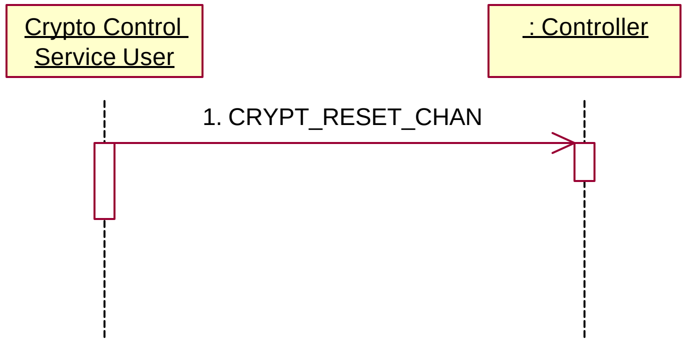

Figure 3-16. Sequence Diagram: Resetting a Crypto Channel using the Crypto Control Service

Figure 3-17 illustrates a Service User invoking the CRYPT_RESET primitive of the Crypto Control Service. The CRYPT_RESET primitive is used to reset an entire cryptographic subsystem.

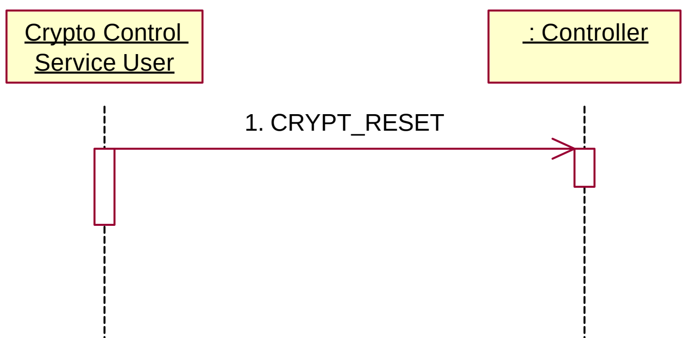

Figure 3-17. Sequence Diagram: Resetting the Cryptographic Subsystem using the Crypto Control Service

3.6.2 Encrypt/Decrypt Service.

The Encrypt/Decrypt services provide for encryption and decryption of data using a given channel created by the Crypto Control Service. The Encrypt/Decrypt primitives can be categorized in Table 3-2.

Table 3-2. Encrypt/Decrypt Primitive Cross-reference Table

CHANNEL TYPE	SERVICE IMPLEMENTATION TYPE	
	MULTIPLE CHANNELS PER OBJECT	**SINGLE CHANNEL PER OBJECT**
Single Sided (Red-Red, Black-Black)	CRYPT_ENCRYPT_WITH_ID, CRYPT_DECRYPT _WITH_ID	CRYPT_ENCRYPT, CRYPT_DECRYPT
Two Sided (Red-Black, Black-Red)	CRYPT_TRANSFORM_WITH_ID	CRYPT_TRANSFROM

The column labels denote the implementation type of the Encrypt/Decrypt Service. Multiple Channels per Object indicates an interface where multiple clients connect to the same server and are multiplexed by channel identifier. Single Channel per Object indicates that each channel has a single client connected to a single server and the channel identifier is implicit. The row labels denote the type of channel created. A single sided channel provides encrypt/decrypt services that return the results back to the Service User (e.g. black side DAMA order wire). For a two-sided channel, the result of the encrypt/decrypt is pushed out of the opposite side of the crypto boundary (e.g. normal data traffic). The Encrypt/Decrypt service provides service primitives to support these types of implementations and channels.

Figure 3-18 illustrates a Service User invoking the CRYPT_ENCRYPT primitive of the Encrypt/Decrypt Service. The CRYPT_ENCRYPT primitive is used to encrypt data and return the data to the Service User. The channel identifier is not passed in the primitive, as it is implicit in the instantiation of the service.

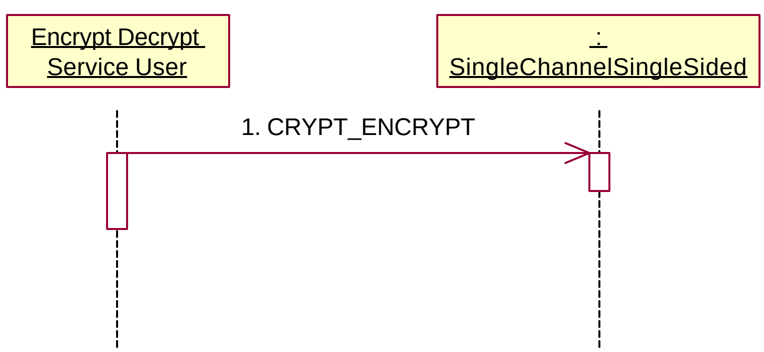

Figure 3-18. Sequence Diagram: Same Side Encryption using the Encrypt/Decrypt Service

Figure 3-19 illustrates a Service User invoking the CRYPT_DECRYPT primitive of the Encrypt/Decrypt Service. The CRYPT_DECRYPT primitive is used to encrypt data and return the data to the Service User. The channel identifier is not passed in the primitive, as it is implicit in the instantiation of the service.

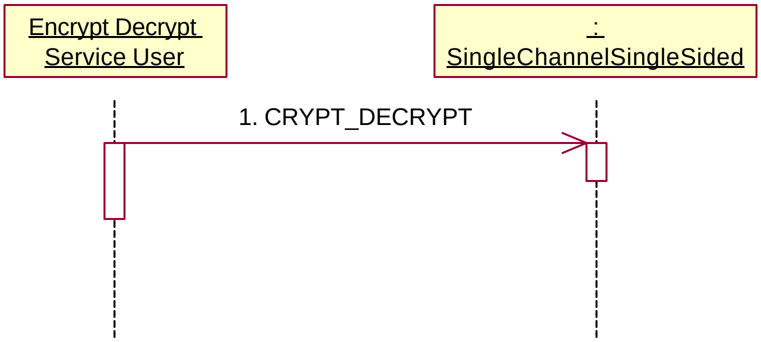

Figure 3-19. Sequence Diagram: Same Side Decryption using the Encrypt Decrypt Service

Figure 3-20 illustrates a Service User invoking the CRYPT_ENCRYPT_WITH_ID primitive of the Encrypt/Decrypt Service. The CRYPT_ENCRYPT_WITH_ID primitive is used to encrypt data and return the data to the Service User. The channel identifier is passed in the primitive.

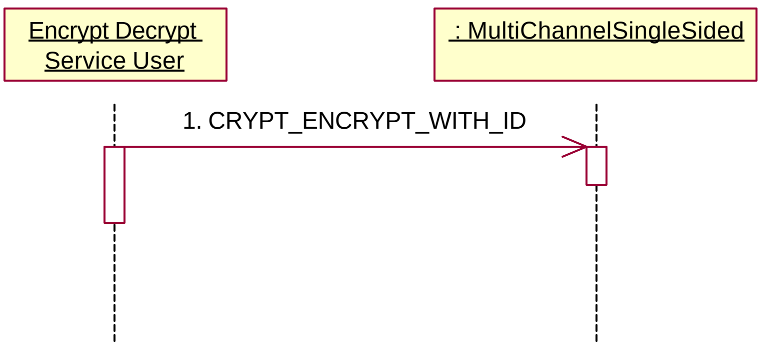

Figure 3-20. Same Side Encryption with Channel Identifier using the Encrypt/Decrypt Service

Figure 3-21 illustrates a Service User invoking the CRYPT_DECRYPT_WITH_ID primitive of the Encrypt/Decrypt Service. The CRYPT_DECRYPT_WITH_ID primitive is used to encrypt data and return the data to the Service User. The channel identifier is passed in the primitive.

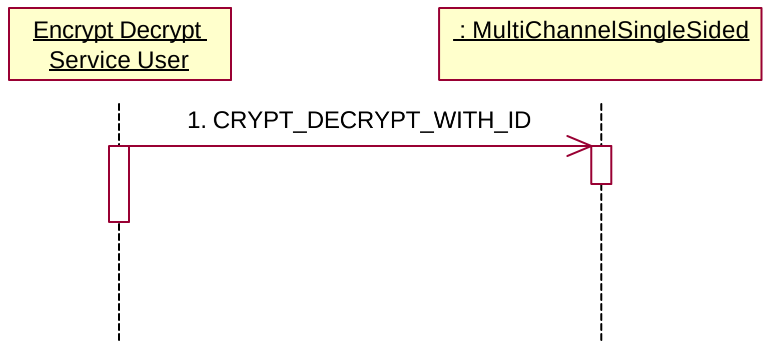

Figure 3-21. Same Side Decryption with Channel Identifier using the Encrypt/Decrypt Service

Figure 3-22 illustrates a Service User invoking the CRYPT_TRANSFORM primitive of the Encrypt/Decrypt Service. The CRYPT_TRANSFORM primitive is used to encrypt/decrypt data. The results of the encryption/decryption appear on the opposite side of the red/black boundary. Header information to bypass the encryption/decryption is provided in the primitive along with the data. The content and size of the header is waveform specific. A corresponding Bypass Policy for the waveform will describe what in the header can be bypassed. The channel identifier is not passed in the primitive, as it is implicit in the instantiation of the service.

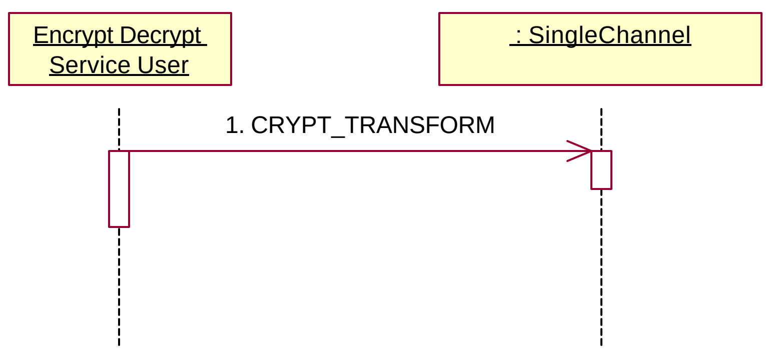

Figure 3-22. Sequence Diagram: Encryption/Decryption using the Encrypt/ Decrypt Service

Figure 3-23 illustrates a Service User invoking the CRYPT_TRANSFORM_WITH_ID primitive of the Encrypt/Decrypt Service. The CRYPT_TRANSFORM_WITH_ID primitive is used to encrypt/decrypt data. This is a multi-channel service. Multiple channels are multiplexed via the channel identifier created by the CRYPT_CREATE_CHAN primitive. This service allows one component to handle multiple instantiated waveform channels. The results of the encryption/decryption appear on the opposite side of the red/black boundary. Header information to bypass the encryption/decryption is provided in the primitive along with the data. The content and size of the header is waveform specific. A corresponding Bypass Policy for the waveform will describe what in the header can be bypassed. The channel identifier is passed in the primitive.

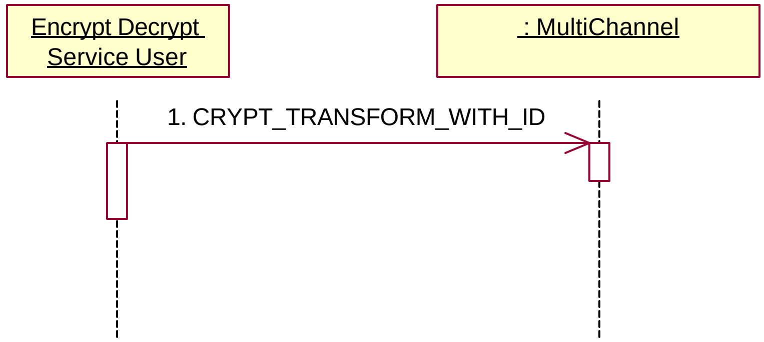

Figure 3-23. Sequence Diagram: Encryption/Decryption with Channel Identifier using the Encrypt/Decrypt Service

3.7 KEY.

Keys require only one service, a management service. Keys in this context are persistent keys, which require storage and are not to be confused with session keys that are generated in a Firefly exchange for example. The Crypto Control Service instantiates traffic channels with the keys managed by the key management service. The logical separation of the service that manages keys and the services that use keys imposes no such separation in the implementation.

3.7.1 Management Service.

The Key Management Service is a specialization of the Fill Management Service with three additional primitives. The KEY_ZEROIZE, KEY_ZEROIZE_ALL, KEY_GET_IDS and KEY_EXPIRY primitives have the same behavior as the corresponding FILL_ZEROIZE, FILL_ZEROIZE_ALL, FILL_GET_IDS and FILL_EXPIRY primitives where the elements are keys.

Figure 3-24 illustrates a Service User invoking the KEY_STORE primitive of the Key Management Service. The KEY_STORE primitive instructs the Key Management service provider to store the current DS-102 fill information as a Key with the name provided in the primitive. Refer to Figure 3-5, which illustrates a complete DS-102 fill sequence using Security Service Primitives.

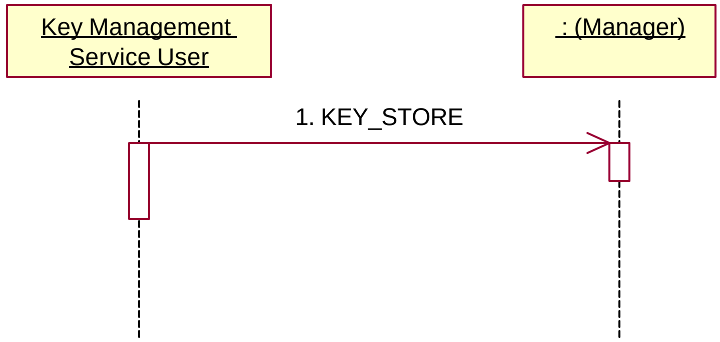

Figure 3-24. Sequence Diagram: Storing a DS-102 Key using the Key Management Service.

Figure 3-25 illustrates a Service User invoking the KEY_UPDATE primitive of the Key Management Service. The KEY_UPDATE primitive instructs the Key Management Service provider to update a specific key that is identified as part of the primitive. The result is a key stored under the same identifier but with an update count incremented by one. Functionally the key is a new key.

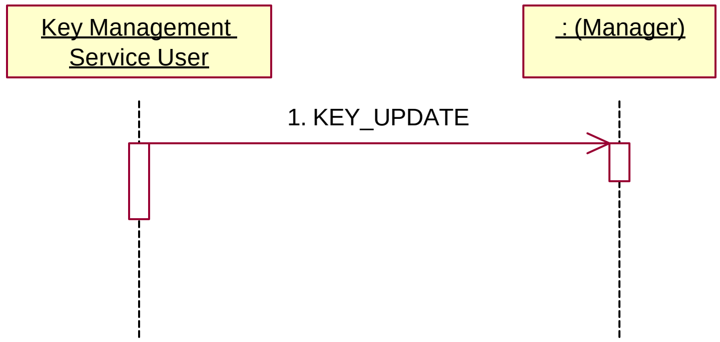

Figure 3-25. Sequence Diagram: Updating a Key using the Key Management Service

Figure 3-26 illustrates a Service User invoking the KEY_GET_UPDATE_COUNT primitive of the Key Management Service. The KEY_GET_UPDATE_COUNT primitive instructs the Key Management Service provider to retrieve the update count of a specific key that is identified as part of the primitive. The count is returned to the user as part of the primitive. The update count is used when coordinating communications between peers to ensure that the key the peers intend to use has the same update count in each radio.

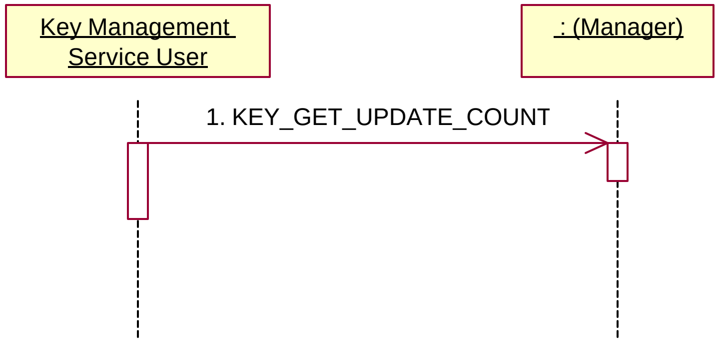

Figure 3-26. Sequence Diagram: Getting the Update Count of a Key using the Key Management Service

3.8 TRANSEC.

TRANSEC requires three services, a management service for managing stored TRANSEC information, a control service for creating and destroying key stream generation channels and a key stream provider service for providing the actual key stream. TRANSEC in this context is persistent information used to generate TRANSEC cover. The Key Stream provider service provides the actual key stream data to a waveform. The logical separation of the TRANSEC management service that manages TRANSEC information and the services that use TRANSEC information imposes no such separation in the implementation.

3.8.1 Control Service.

The TRANSEC Control Service instantiates and destroys classified key stream generation channels with the TRANSEC information managed by the TRANSEC management service.

Figure 3-27 illustrates a Service User invoking the TRAN_CREATE_CHAN primitive of the TRANSEC Control Service. The TRAN_CREATE_CHAN primitive causes the Control Service to create a classified key stream generation channel. The TRANSEC algorithm, key and seed are specified as part of the primitive. An opaque channel identifier is returned to the Service User.

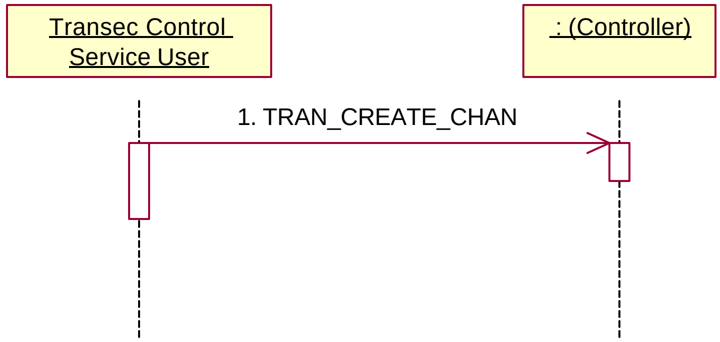

Figure 3-27. Sequence Diagram: Creating a TRANSEC Channel (Key Stream) using the TRANSEC Control Service

Figure 3-28 illustrates a Service User invoking the TRAN_GET_CHAN_CONFIG primitive of the TRANSEC Control Service. The TRAN_GET_CHAN_CONFIG primitive causes the Control Service to return the configuration of a key stream generation channel. The TRANSEC channel identifier is input as part of the primitive. The configuration information that was used to create the channel is returned to the Service User.

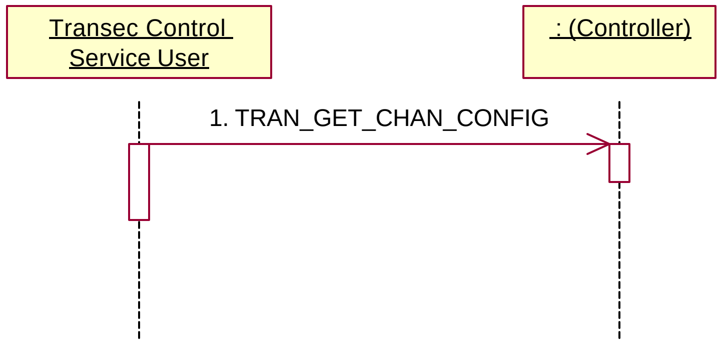

Figure 3-28. Sequence Diagram: Getting a TRANSEC Channel Configuration using the TRANSEC Control Service

Figure 3-29 illustrates a Service User invoking the TRAN_DESTROY_CHAN primitive of the TRANSEC Control Service. The TRAN_DESTROY_CHAN primitive causes the Control Service to destroy a key stream generation channel. The TRANSEC channel identifier is input as part of the primitive.

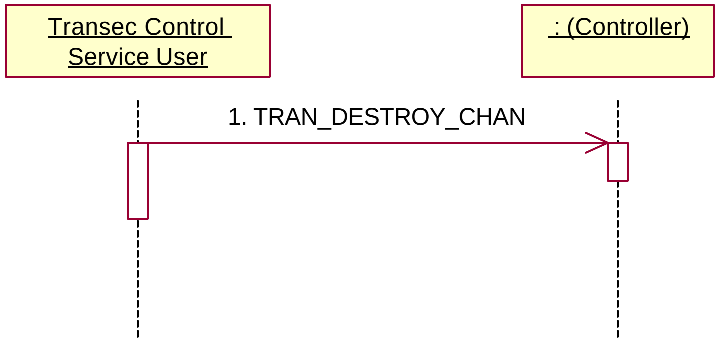

Figure 3-29. Sequence Diagram: Destroying a TRANSEC Channel using the TRANSEC Control Service

3.8.2 Key Stream Service.

The Key Stream Service provides generated classified key stream data from a channel instantiated by the TRANSEC Control Service.

Figure 3-30 illustrates a Service User invoking the TRAN_GEN_KEY_STREAM primitive of the TRANSEC Key Stream Service. The TRAN_GEN_KEY_STREAM generates a classified key stream based on the algorithm and key provided to the TRAN_CREATE_CHAN primitive. A new seed is provided as input to this primitive. The resulting key stream is returned to the Service User as part of the primitive. The channel identifier is input as part of the primitive (multi-channel service).

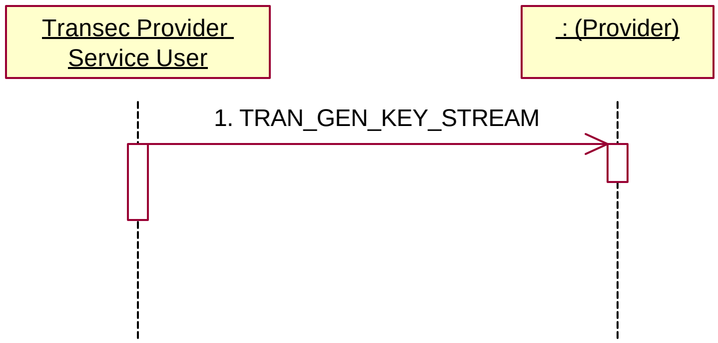

Figure 3-30. Sequence Diagram: Generating a Key Stream with a New Seed using the TRANSEC Key Stream Service.

Figure 3-31 illustrates a Service User invoking the TRAN_GEN_NEXT_KEY_STREAM primitive of the TRANSEC Key Stream Service. The TRAN_GEN_KEY_STREAM is identical to the TRAN_GEN_KEY_STREAM primitive except that a new seed is not provided and the key stream is generated based on the existing state of the channel.

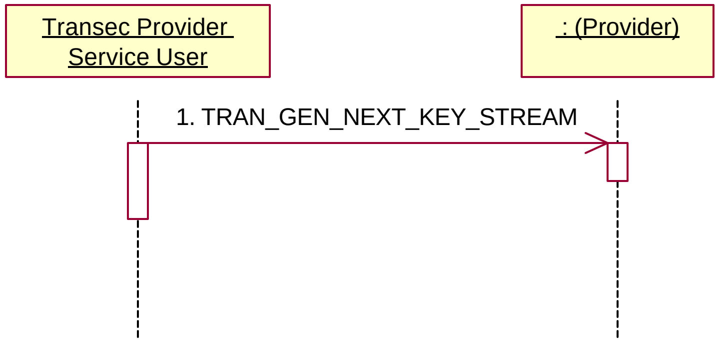

Figure 3-31. Sequence Diagram: Generating a Key Stream without a New Seed using the TRANSEC Key Stream Service.

3.8.3 Management Service.

The TRANSEC Management Service is a specialization of the Fill Management Service with two additional primitives. The TRAN_ZEROIZE, TRAN _ZEROIZE_ALL, TRAN _GET_IDS and TRAN_EXPIRY primitives have the same behavior as the corresponding FILL_ZEROIZE, FILL_ZEROIZE_ALL, FILL_GET_IDS and FILL_EXPIRY primitives where the elements are TRANSEC information.

Figure 3-32 illustrates a Service User invoking the TRAN_STORE primitive of the TRANSEC Management Service. The TRAN_STORE primitive instructs the TRANSEC management service provider to store the current DS-102 fill information as TRANSEC information with the name provided in the primitive.

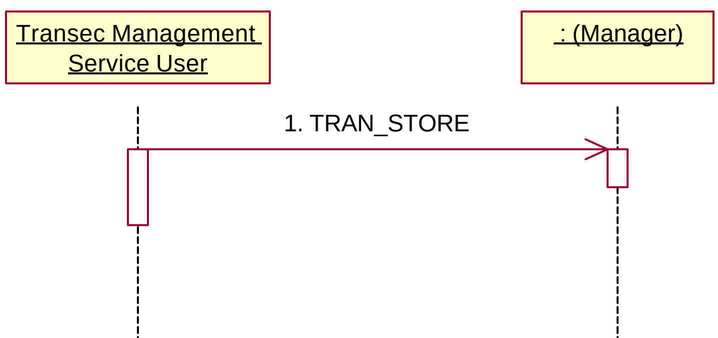

Figure 3-32. Sequence Diagram: Storing DS-102 TRANSEC Information using the TRANSEC Management Service

Figure 3-33 illustrates a Service User invoking the TRAN_GET_FILL primitive of the TRANSEC Management Service. The TRAN_GET_FILL primitive instructs the TRANSEC Management service provider to return the unclassified fill information associated with the identifier provided in the primitive to the service user. Refer to Figure 3-5, which illustrates a complete DS-102 fill sequence using Security Service Primitives.

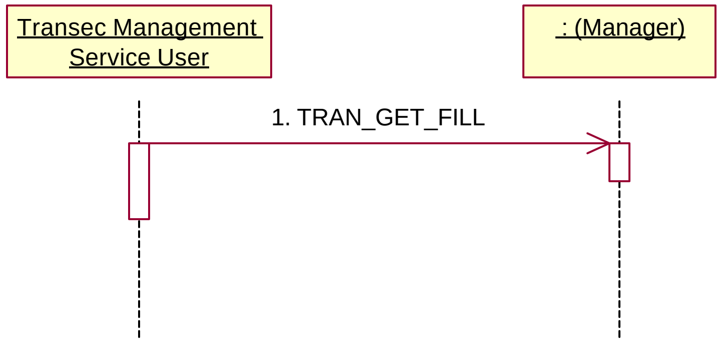

Figure 3-33. Sequence Diagram: Getting Unclassified TRANSEC Fill Info using the TRANSEC Management Service

3.9 POLICY.

Policies in the context of the JTRS Security Service API are information used to control the behavior of the JTRS Security Enforcement mechanisms. The number and content of the policies in any given JTRS platform will vary according to the platform configuration and the number and type of waveform applications loaded on it. Policies are used to parameterize crypto bypass behavior, access control to objects and files, and audit behavior. Figure 3-34 illustrates how bypass policies are used. The Control Bypass Guard enforces System and Waveform configuration and control bypass policies that are accessed from a Policy store. The Bypass policies contain information that the guard uses in its enforcement mechanism to either allow or disallow information to flow from red to black. The Header Bypass Guard is similar except that it performs its enforcement function at real time data rates and on header information that is associated with packets of data.

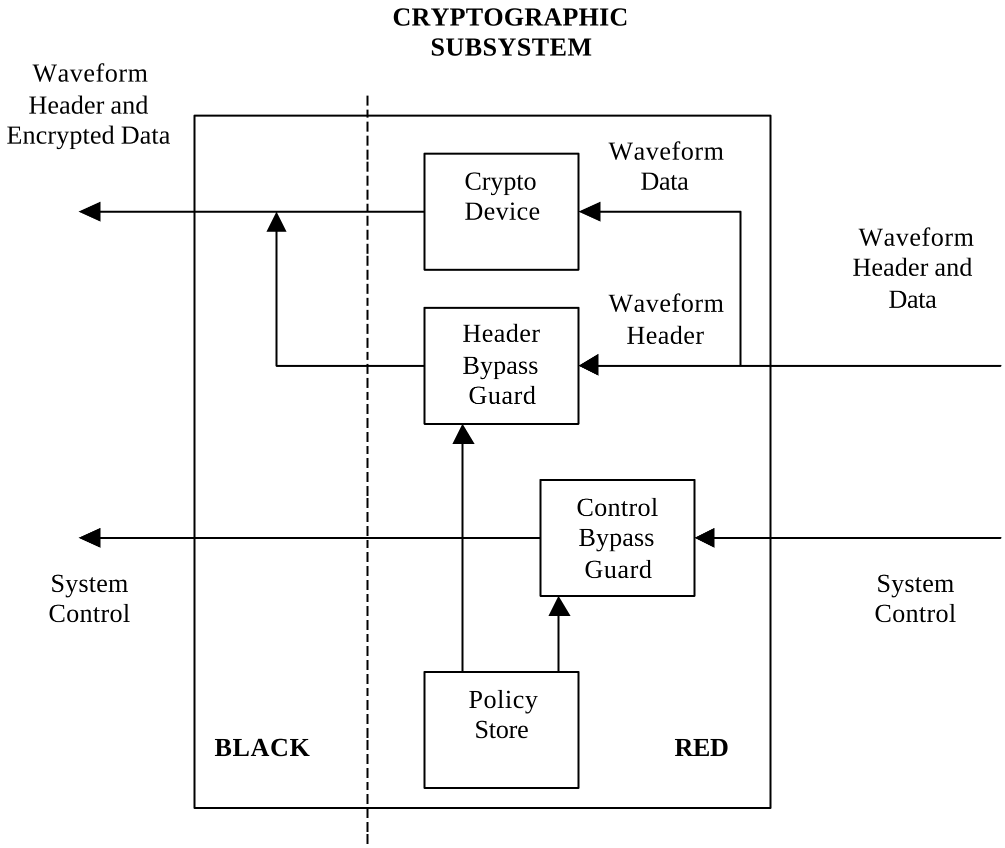

Figure 3-34. Security Policies and Bypass

3.9.1 Management Service.

The Policy Management Service is a specialization of the Fill Management Service with one additional primitive. The POL_ZEROIZE, POL_ZEROIZE_ALL, POL_GET_IDS and POL_EXPIRY primitives have the same behavior as the corresponding FILL_ZEROIZE, FILL_ZEROIZE_ALL, FILL_GET_IDS and FILL_EXPIRY primitives where the elements are policies.

Figure 3-35 illustrates a Service User invoking the POL_GET_POLICY primitive of the Policy Management Service. The POL_GET_POLICY primitive instructs the Policy Management Service provider to return the security policy associated with the identifier provided in the primitive to the service user. The Policy Management Service provider returns the policy in the same primitive.

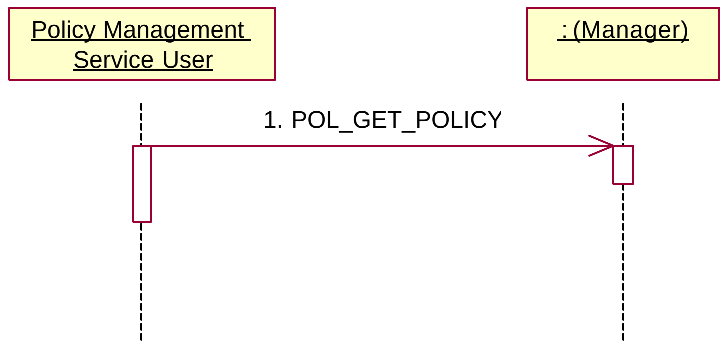

Figure 3-35. Sequence Diagram: Getting a Security Policy using the Policy Management Service.

3.10 INTEGRITY AND AUTHENTICATION.

Integrity and Authentication encompasses verification of the identity of the source of information (authentication) and verification that the information has not been changed (integrity). Certificates are used to generate the Integrity and Authentication context.

3.10.1 Control and Digital Signatures Provider Services.

Figure 3-36 illustrates the sequence of primitives to digitally sign a file.

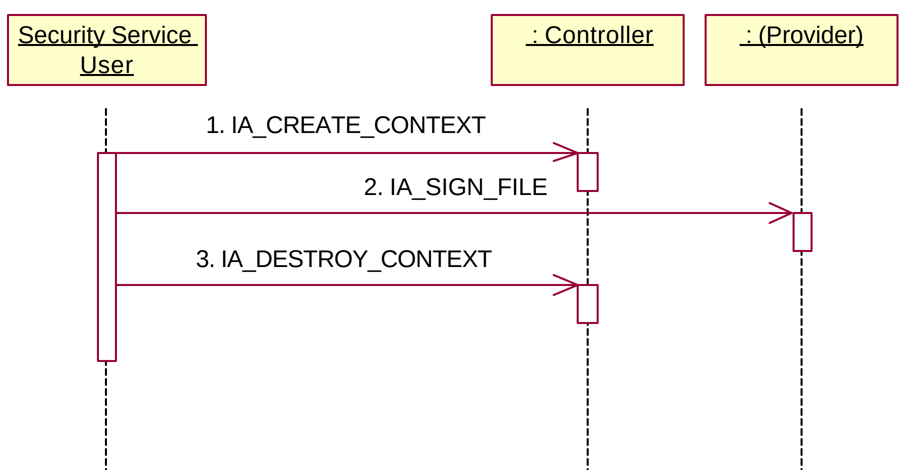

Figure 3-36. Sequence Diagram: Signing a File

1. The Service User invokes the IA_CREATE_CONTEXT primitive of the Control Service. The identifier of the certificate to use to create the context is supplied as part of the primitive. The certificate identifies the hashing and signature algorithms to be used.
2. The Service User invokes the IA_SIGN_FILE primitive of the Digital Signatures Provider Service to sign a file.

3. The Service User invokes the IA_DESTROY_CONTEXT context primitive of the Digital Signatures Provider Service to destroy the context.

Figure 3-37 illustrates the sequence of primitives to digitally verify a file.

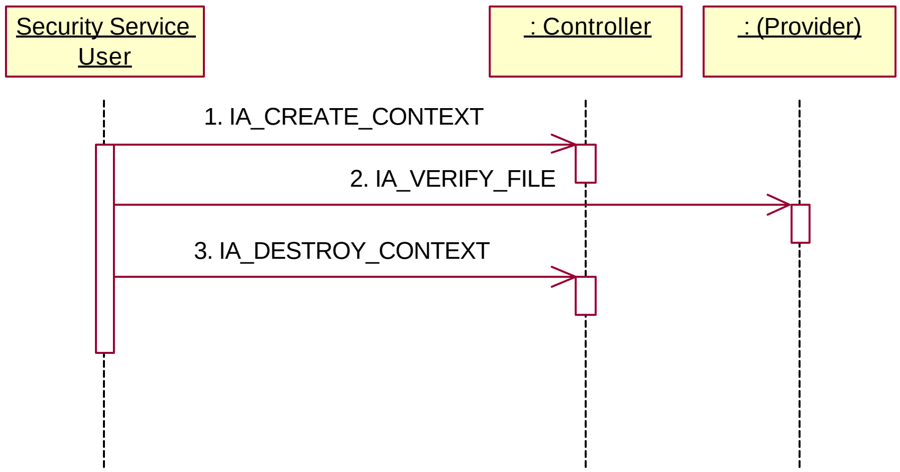

Figure 3-37. Sequence Diagram: Verifying a File

1. The Service User invokes the IA_CREATE_CONTEXT primitive of the Control Service. The identifier of the certificate to use to create the context is supplied as part of the primitive.
2. The Service User invokes the IA_VERIFY_FILE primitive of the Digital Signatures Provider Service to verify a digitally signed file. The result of the verification is returned in the primitive.
3. The Service User invokes the IA_DESTROY_CONTEXT context primitive of the Digital Signatures Provider Service to destroy the context.

Figure 3-38 illustrates the sequence of primitives to generate and sign a hash.

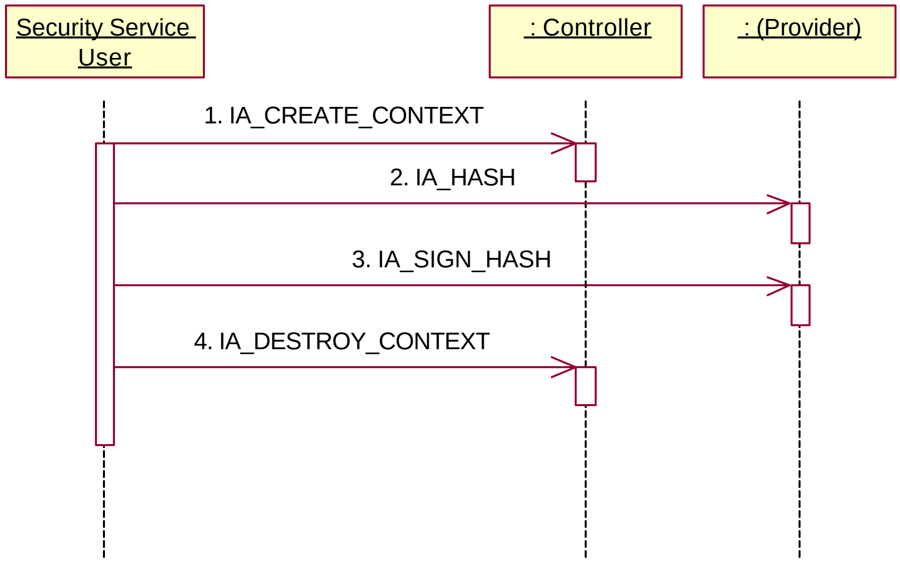

Figure 3-38. Sequence Diagram: Generating and Signing a Hash.

1. The Service User invokes the IA_CREATE_CONTEXT primitive of the Control Service. The identifier of the certificate to use to create the context is supplied as part of the primitive. An internal hash is initialized as part of the context.
2. The Service User invokes the IA_HASH primitive of the Digital Signatures Provider Service to update the internal hash from data supplied with the primitive. This primitive may executed as many times in succession as required to generate the required hash (e.g. a block of data must be broken up into 2 or more pieces for reasons of time or size). Once IA_HASH is invoked for a context it is invalid to invoke the IA_SIGN_FILE or IA_VERIFY_FILE primitives, as they would invalidate the hash.
3. The Service User invokes the IA_SIGN_HASH primitive to sign the generated hash. The resultant digital signature is returned in the primitive.
4. The Service User invokes the IA_DESTROY_CONTEXT context primitive of the Digital Signatures Provider Service to destroy the context.

Figure 3-38 illustrates the sequence of primitives to verify a digitally signed block of data.

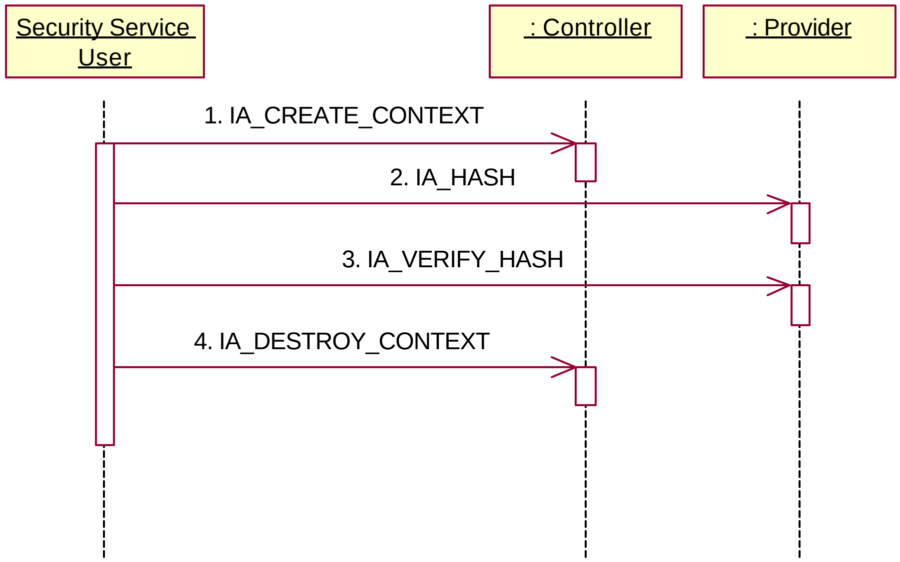

Figure 3-39. Sequence Diagram: Verifying a Digital Signature

1. The Service User invokes the IA_CREATE_CONTEXT primitive of the Control Service. The identifier of the certificate to use to create the context is supplied as part of the primitive. An internal hash is initialized as part of the context.
2. The Service User invokes the IA_HASH primitive of the Digital Signatures Provider Service to update the internal hash from data supplied with the primitive. This primitive may executed as many times in succession as required to generate the required hash. Once IA_HASH is invoked for a context it is invalid to invoke the IA_SIGN_FILE or IA_VERIFY_FILE primitives, as they will invalidate the hash.
3. The Service User invokes the IA_VERIFY_HASH primitive to verify the digital signature supplied with the data matches the generated hash. The resultant digital signature is returned in the primitive.
4. The Service User invokes the IA_DESTROY_CONTEXT context primitive of the Digital Signatures Provider Service to destroy the context.

3.11 ALARM.

The Security Service divides alarms into two components, an audit record and an alarm indicator. The audit record is modeled after the ITU X.736 standard. The CF::*Logger* is used to log the audit record.

Figure 3-40 illustrates a Security Service Provider issuing the ALARM_SIGNAL primitive. This primitive signals the Security Service User that a crypto alarm has occurred.

3.11.1 User.

Figure 3-40 illustrates a Security Service Provider issuing the ALARM_SIGNAL primitive. This primitive signals the Security Service User that a crypto alarm has occurred.

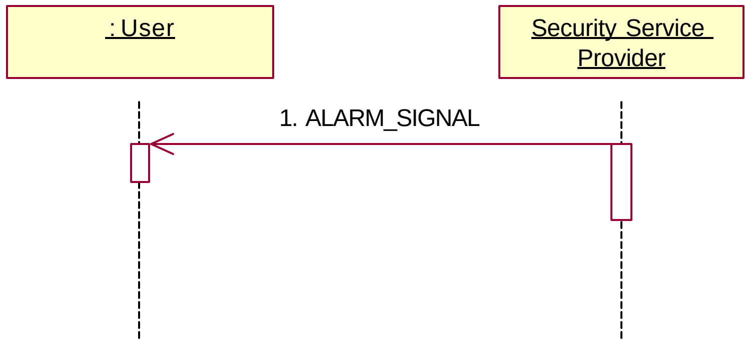

Figure 3-40. Sequence Diagram: Signaling a Crypto Alarm

3.12 TIME.

Some Security Service implementations require management of time. The Security Service API defines a Time Management Service for this purpose.

3.12.1 Management Service.

Figure 3-41 illustrates a Service User invoking the TIME_SET_TIME primitive of the Time Management Service. TIME_SET_TIME primitive is used to set the time of day in for the Security Service.

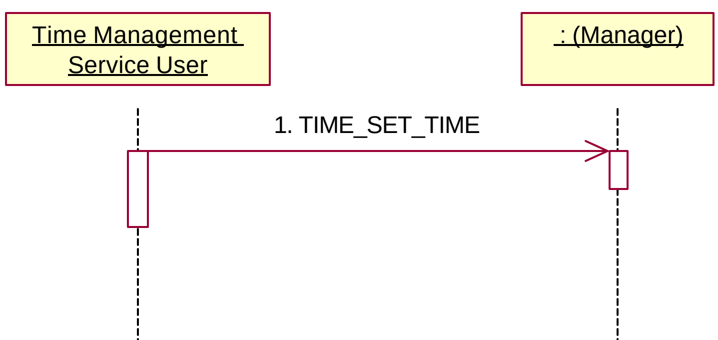

Figure 3-41. Sequence Diagram: Setting Time using the Time Management Service

Figure 3-42 illustrates a Service User invoking the TIME_GET_TIME primitive of the Time Management Service. TIME_GET_TIME primitive is used to request the time of day maintained within the Security Service. The time of day is returned in the primitive.

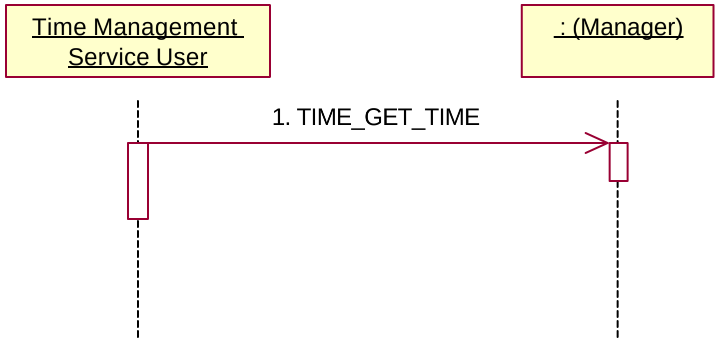

Figure 3-42. Sequence Diagram: Getting Time using the Time Management Service

Figure 3-43 illustrates a Service User invoking the TIME_SET_DATE primitive of the Time Management Service. TIME_SET_DATE primitive is used to set the date in the Security Service.

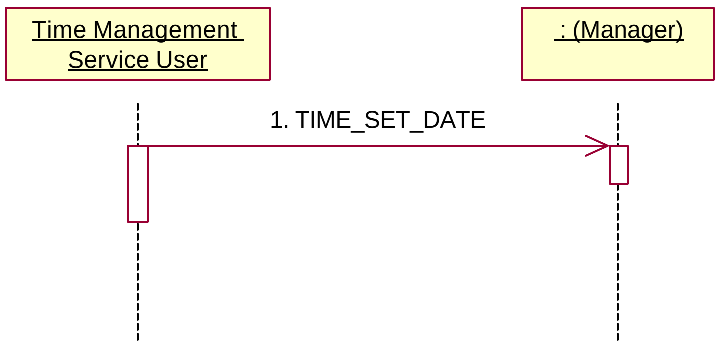

Figure 3-43. Sequence Diagram: Setting Date using the Time Management Service

Figure 3-43 illustrates a Service User invoking the TIME_GET_DATE primitive of the Time Management Service. TIME_GET_DATE primitive is used to get the date maintained in the Security Service.

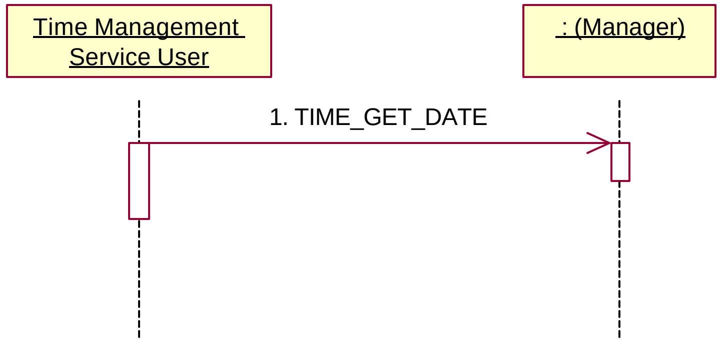

Figure 3-44. Sequence Diagram: Getting Date using the Time Management Service

3.13 GPS.

3.13.1 Management.

The GPS Management Service is a specialization of the TRANSEC Management Service with no additional primitives. The GPS_ZEROIZE, GPS_ZEROIZE_ALL, GPS_GET_IDS, GPS_EXPIRY, GPS_STORE and GPS_GET_FILL primitives have the same behavior as the corresponding TRAN_ZEROIZE, TRAN_ZEROIZE_ALL, TRAN_GET_IDS TRAN_EXPIRY, TRAN_STORE and TRAN_GET_FILL primitives.

4 SERVICE PRIMITIVES.

The entirety of the JTRS Security API set is defined within a CORBA module called JTRSSecurity. There are common types that are used by multiple modules within the JTRSSecurity module. They are shown in Figure 4-1.

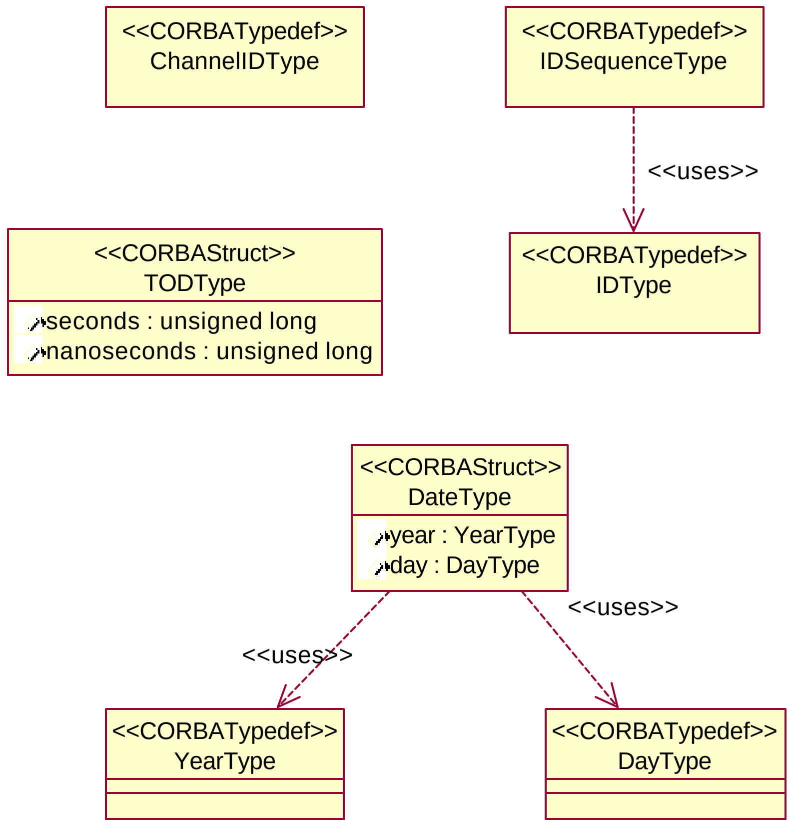

Figure 4-1. Class Diagram: JTRS Security Common Types

The service primitives, as shown in Table 3-1 are broken up into service groups. Each service group translates into a CORBA module within the JTRSSecurity module and each service within the group translates into an interface. This organization provides scope for names. For example the full scoped name of the Key Management Service is JTRSSecurity::Key::Manager.

4.1 SECURITY.

There is a management service that exists at the JTRSSecurity level and is shown in Figure 4-2.

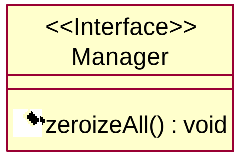

Figure 4-2. Class Diagram: JTRS Security Management Service

4.1.1 ZEROIZE_ALL.

The FILL_ZEROIZE_ALL primitive deletes all managed elements within the entirety of the Security Service. These elements are algorithms, keys, TRANSEC certificates, Policies and GPS.

4.1.1.1 Synopsis.

void zeroizeAll () raises (ZeroizeFailed);

4.1.1.2 Parameters.

N/A.

4.1.1.3 State.

This primitive is valid in any state.

4.1.1.4 New State.

The resulting state is unchanged.

4.1.1.5 Response.

N/A.

4.1.1.6 Originator.

This primitive is initiated by the service user.

4.1.1.7 Errors/Exceptions.

ZeroizeFailed

The zeroize failed for an indeterminate reason.

4.2 FILL.

The Fill services are shown in Figure 4-3. These services support filling a cryptographic subsystem through a Fill Port (Port, PortUser), filling through file input (Bus) and management of store fill information (Manager).

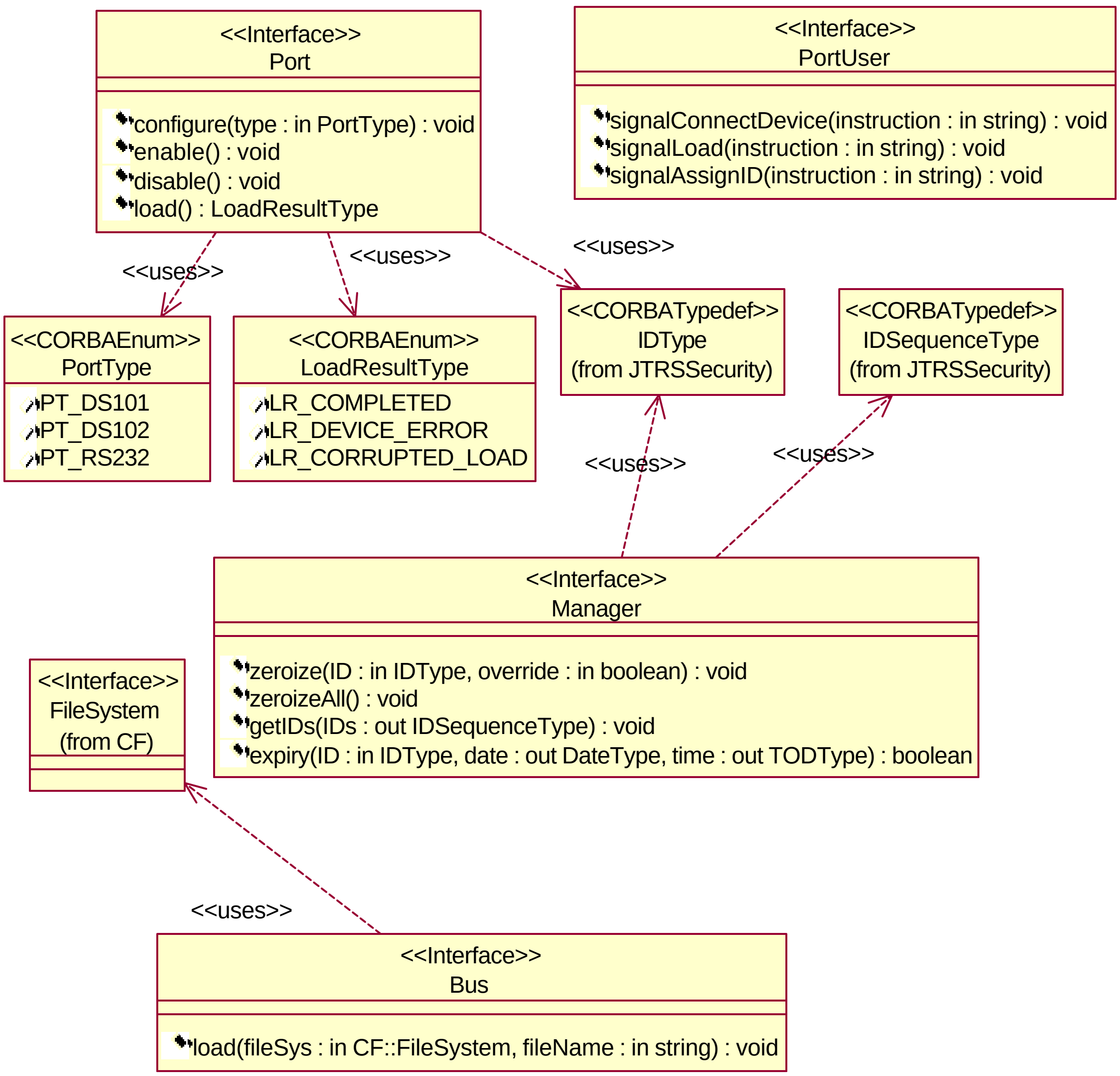

Figure 4-3. Class Diagram: Fill Services

4.2.1 FILL_PORT_CONFIGURE.

This primitive configures the Fill Port for DS-101, DS-102 or RS-232 operation.

4.2.1.1 Synopsis.

```
void configure (
        in PortType     type
);
```

4.2.1.2 Parameters.

type

Indicates how to configure the Fill Port

PT_DS101	Configure the Fill Port for DS101 operation
PT_DS102	Configure the Fill Port DS102 operation
PT_RS232	Configure the Fill Port RS-232 operation

4.2.1.3 State.

This primitive is valid in the DISABLED state.

4.2.1.4 New State.

The state remains unchanged.

4.2.1.5 Response.

The FILL_PORT_SIGNAL_CONNECT primitive is invoked on the service user.

4.2.1.6 Originator.

This primitive is initiated by the service user.

4.2.1.7 Errors/Exceptions.

N/A.

4.2.2 FILL_PORT_ENABLE.

This primitive enables the Fill Port. The port will be enabled with the configuration set by the FILL_PORT_CONFIGURE primitive.

4.2.2.1 Synopsis.

void enable ();

4.2.2.2 Parameters.

N/A.

4.2.2.3 State.

This primitive is valid in the DISABLED state.

4.2.2.4 New State.

The new state is ENABLED.

4.2.2.5 Response.

The FILL_PORT_SIGNAL_LOAD primitive is invoked on the service user.

4.2.2.6 Originator.

This primitive is initiated by the service user.

4.2.2.7 Errors/Exceptions.

N/A.

4.2.3 FILL_PORT_DISABLE.

This primitive disables the Fill Port.

4.2.3.1 Synopsis.

void disable ();

4.2.3.2 Parameters.

N/A.

4.2.3.3 State.

This primitive is valid in the ENABLED state.

4.2.3.4 New State.

The resulting state is DISABLED.

4.2.3.5 Response.

N/A.

4.2.3.6 Originator.

This primitive is initiated by the service user.

4.2.3.7 Errors/Exceptions.

N/A.

4.2.4 FILL_PORT_LOAD.

This primitive initiates the load of fill information from the fill device. The primitive returns to the caller when the load terminates.

4.2.4.1 Synopsis.

LoadResultType load ();

4.2.4.2 Parameters.

N/A.

4.2.4.3 State.

This primitive is valid in the ENABLED state.

4.2.4.4 New State.

A transition to the LOAD_IN_PROGRESS state is made upon entering this primitive. Upon termination a return to the ENABLED state is made.

4.2.4.5 Response.

N/A.

4.2.4.6 Originator.

This primitive is initiated by the service user.

4.2.4.7 Errors/Exceptions.

The primitive returns a status:

LR_SUCCESS	The load completed successfully.
LR_DEVICE_ERROR	A device error occurred. The fill device may not be connected or may be faulty.
LR_CORRUPTED_LOAD	The loaded data is corrupt.

4.2.5 FILL_PORT_SIGNAL_CONNECT.

This primitive signals the service user to connect the Fill Device to the Fill Port.

4.2.5.1 Synopsis.

```
void signalConnectDevice (
        in string        instruction
);
```

4.2.5.2 Parameters.

string

Provides any additional information to the user about connecting the device.

4.2.5.3 State.

This primitive is issued in the DISABLED state.

4.2.5.4 New State.

The state remains unchanged.

4.2.5.5 Response.

N/A.

4.2.5.6 Originator.

This primitive is initiated by the service provider.

4.2.5.7 Errors/Exceptions.

N/A.

4.2.6 FILL_PORT_SIGNAL_LOAD.

This primitive signals the Service User that the Fill Port is ready for the Service User to initiate the load.

4.2.6.1 Synopsis.

```
void signalLoad (
        in string         instruction
);
```

4.2.6.2 Parameters.

instruction

A string which may contain additional instructions for initiating the load.

4.2.6.3 State.

This primitive is issued from the ENABLED state.

4.2.6.4 New State.

The state remains unchanged.

4.2.6.5 Response.

N/A.

4.2.6.6 Originator.

This primitive is initiated by the service provider.

4.2.6.7 Errors/Exceptions.

N/A.

4.2.7 FILL_PORT_SIGNAL_ASSIGN_ID.

This primitive signals the Service User that the appropriate Fill Manager (TRANSEC or KEY) is ready for the Service User to initiate assign an ID to the fill information.

4.2.7.1 Synopsis.

```
void signalAssignId (
        in string        instruction
);
```

4.2.7.2 Parameters.

instruction

A string that may contain additional instructions for assigning an ID to the fill information loaded via a DS-102 fill device.

4.2.7.3 State.

This primitive is issued from the FILL_STATE_PENDING_STORE state.

4.2.7.4 New State.

The state remains unchanged.

4.2.7.5 Response.

N/A.

4.2.7.6 Originator.

This primitive is initiated by the service provider.

4.2.7.7 Errors/Exceptions.

N/A.

4.2.8 FILL_BUS_LOAD.

This primitive loads fill information that is stored in a file. The file may or may not be encrypted but will be digitally signed. All keys and cryptographic algorithms will be encrypted and digitally signed.

4.2.8.1 Synopsis.

```
void load (
        in CF::FileSystem      fileSys,
        in string              fileName
);
```

4.2.8.2 Parameters.

fileSys

Identifies the location of the file of fill information.

fileName

The name of the file.

4.2.8.3 State.

This primitive may be issued in any state.

4.2.8.4 New State.

The state remains unchanged.

4.2.8.5 Response.

N/A.

4.2.8.6 Originator.

This primitive is initiated by the service user.

4.2.8.7 Errors/Exceptions.

This primitive may raise the exceptions associated with the CF::*FileSystem* and CF::*File*. In addition the following exception may be raised:

FileNotValid

The file is not a valid fill file.

4.2.9 FILL_ZEROIZE.

This primitive deletes all instances of a single element from a security service as specified by the ID.

4.2.9.1 Synopsis.

```
void zeroize (
        in IdType       id
        in boolean      override
) raises (InvalidId, ElementInUse, ZeroizeFailed);
```

4.2.9.2 Parameters.

id

Identifies the element to delete

override

Causes the element to be deleted even though the element is in use by an instantiated channel or context. If the element is in use, then all processing using the element must be terminated.

4.2.9.3 State.

The primitive is valid in any state.

4.2.9.4 New State.

The resulting state is unchanged.

4.2.9.5 Response.

N/A.

4.2.9.6 Originator.

This primitive is initiated by the service user.

4.2.9.7 Errors/Exceptions

The following exceptions may be raised:

InvalidId

The id is either malformed or the element does not exist.

ElementInUse

The element to be removed is currently in use by an instantiated channel or context. This exception only is raised if the override parameter is set to FALSE.

ZeroizeFailed

The zeroize failed for an indeterminate reason.

4.2.10 FILL_ZEROIZE_ALL.

The FILL_ZEROIZE_ALL primitive deletes all elements of a given type from a security service. Any processing using the elements is terminated.

4.2.10.1 Synopsis.

void zeroizeAll () raises (ZeroizeFailed);

4.2.10.2 Parameters.

N/A.

4.2.10.3 State.

This primitive is valid in any state.

4.2.10.4 New State.

The resulting state is unchanged.

4.2.10.5 Response.

N/A.

4.2.10.6 Originator.

This primitive is initiated by the service user.

4.2.10.7 Errors/Exceptions.

ZeroizeFailed

The zeroize failed for an indeterminate reason.

4.2.11 FILL_GET_IDS.

This primitive retrieves the identifiers of all the elements associated with the manager that are resident in a security system (e.g. keys for Key::Manager)

4.2.11.1 Synopsis.

```
void getIds (
        out IdSequenceType   ids
);
```

4.2.11.2 Parameters.

ids

> A sequence of identifiers of all the elements associated with the manager in the security subsystem. The number of elements is implicit in the sequence.

4.2.11.3 State.

This primitive is valid in any state.

4.2.11.4 New State.

The resulting state is unchanged.

4.2.11.5 Response.

N/A.

4.2.11.6 Originator.

This primitive is initiated by the service user.

4.2.11.7 Errors/Exceptions.

N/A.

4.2.12 FILL_EXPIRY.

This primitive retrieves the expiration date and time for a given element associated with a manager that is resident in a security system (e.g. Certificate for Certificate::Manager).

4.2.12.1 Synopsis.

```
boolean expiry (
        in IdType       id,
        out DateType    date,
        out TODType     time
) raises (InvalidId);
```

4.2.12.2 Parameters.

id

The ID of the element for which to retrieve the expiration time and date.

date

The expiration date of the element. See paragraph 4.11.3.2 for the structure of DateType.

time

The expiration time of the element. See paragraph 4.11.1.2 for the structure of TODType.

4.2.12.3 State.

This primitive is valid in any state.

4.2.12.4 New State.

The resulting state is unchanged.

4.2.12.5 Response.

This primitive returns a boolean:

FALSE   The element does not expire.

TRUE   The element expires at the time and date returned from the primitive.

4.2.12.6 Originator.

This primitive is initiated by the service user.

4.2.12.7 Errors/Exceptions.

The following exception is raised:

InvalidId

The id is either malformed or the element does not exist.

4.3 ALGORITHM.

The Algorithm Management Service is a specialization of the Fill Management Service as shown in Figure 4-4. It is responsible for management of stored COMSEC and TRANSEC algorithms.

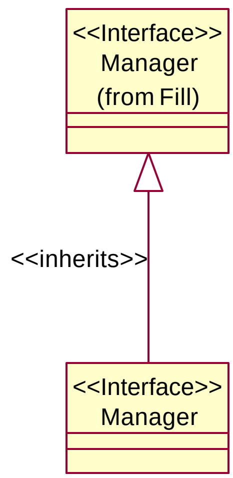

Figure 4-4. Class Diagram: Algorithm Management Service

4.3.1 ALG_ZEROIZE.

This primitive deletes all instances of a single cryptographic algorithm from a security service as specified by the id. See paragraph 4.2.9 for the semantics and behavior.

4.3.2 ALG_ZEROIZE_ALL.

This primitive deletes cryptographic algorithms from a security service. See paragraph 4.2.10 for the semantics and behavior.

4.3.3 ALG_GET_IDS.

This primitive retrieves the identifiers of all the cryptographic algorithms resident in a security service. See paragraph 4.2.11 for the semantics and behavior.

4.3.4 ALG_EXPIRY.

This primitive retrieves the expiration date and time for a given algorithm within a security service. See paragraph 4.2.12 for the semantics and behavior.

4.4 CERTIFICATE.

The Certificate Management Service is a specialization of the Fill Management Service as shown in Figure 4-4. It is responsible for management of digital certificates which support the Integrity and Authentication services.

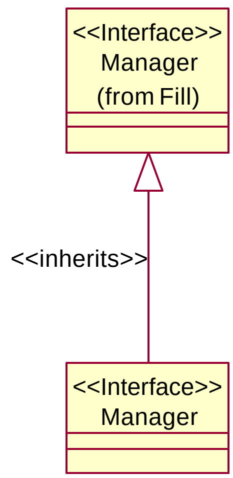

Figure 4-5. Class Diagram: Certificate Management Service

4.4.1 CERT_ZEROIZE.

This primitive deletes all instances of a single certificate from a security service as specified by the id. See paragraph 4.2.9 for the semantics and behavior.

4.4.2 CERT_ZEROIZE_ALL.

This primitive deletes all certificates from a security service. See paragraph 4.2.10 for the semantics and behavior.

4.4.3 CERT_GET_IDS.

This primitive retrieves the identifiers of all the certificates resident in a security service. See paragraph 4.2.11 for the semantics and behavior.

4.4.4 CERT_EXPIRY.

This primitive retrieves the expiration date and time for a given certificate within a security service. See paragraph 4.2.12 for the semantics and behavior.

4.5 CRYPTO.

Figure 4-6 shows the class diagram of the Crypto Control Service. This interface supports the instantiation, tear down and basic mode control for a crypto channel.

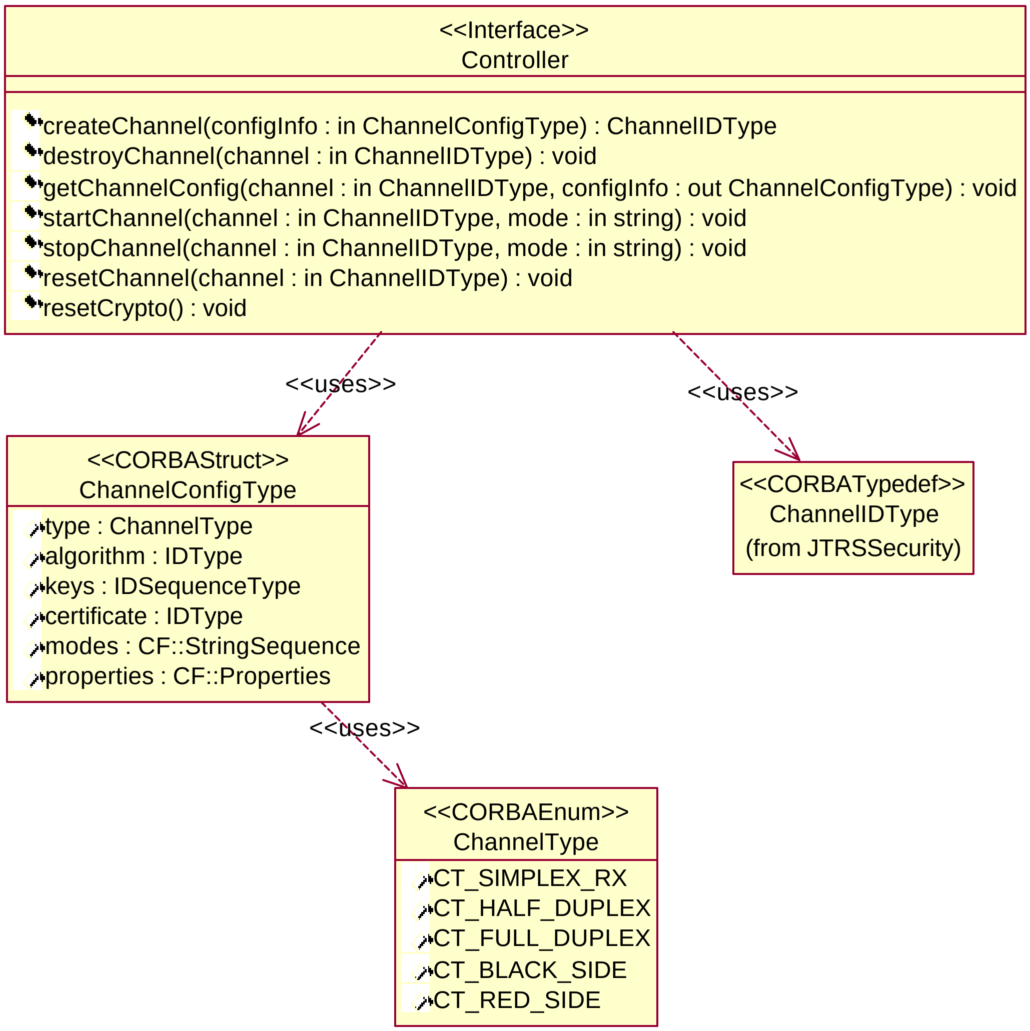

Figure 4-6. Class Diagram: Crypto Control Service

Figure 4-7 shows the Encrypt/Decrypt services. These services provide the ability to encrypt and decrypt between the red and black sides of a radio. In addition red-red and black-black encrypt/decrypt services are provided to support cases such as DAMA order wire.

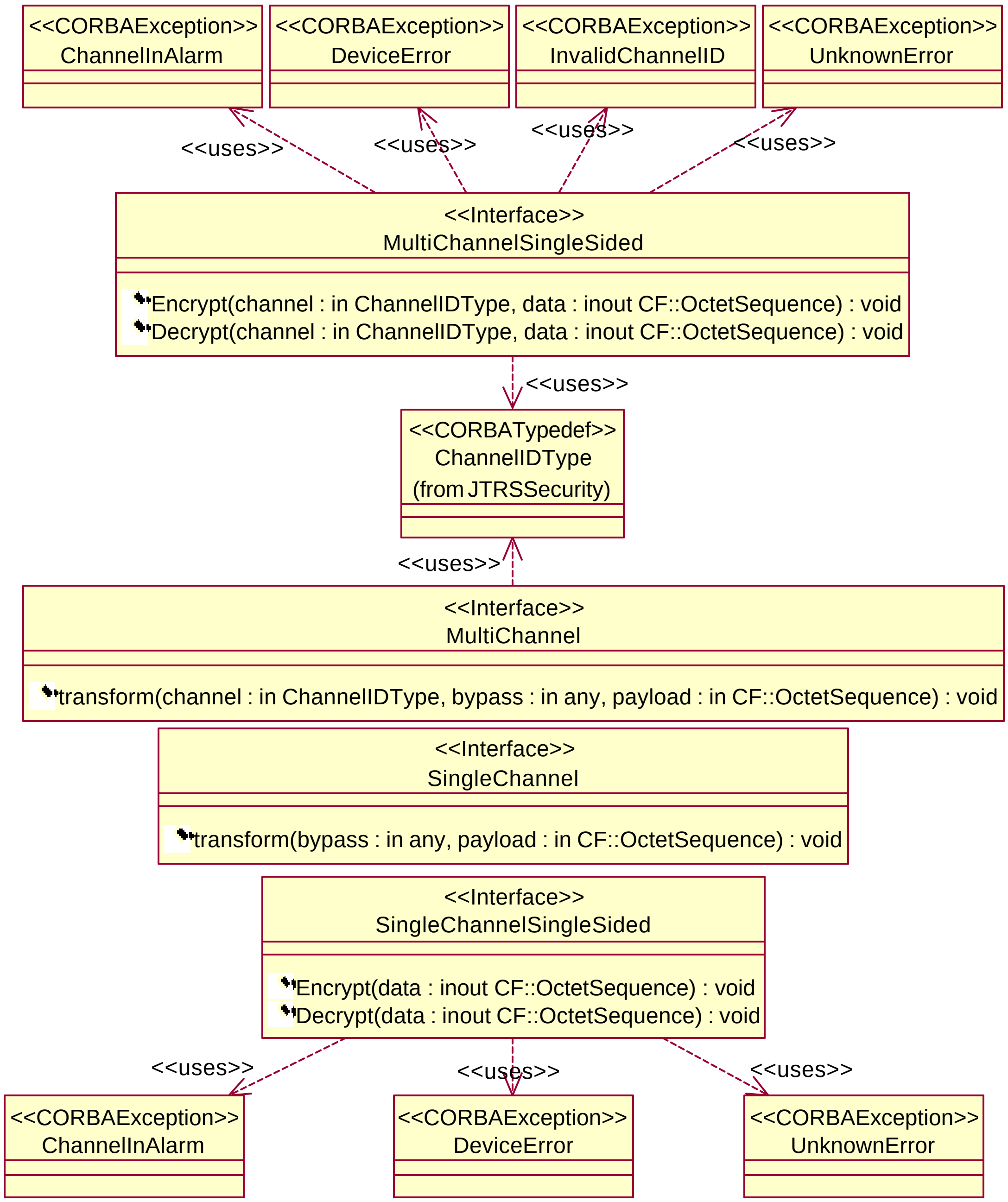

Figure 4-7. Class Diagram: Encrypt/Decrypt Services

4.5.1 CRYPT_CREATE_CHAN.

This primitive creates a COMSEC channel within a cryptographic subsystem.

4.5.1.1 Synopsis.

```
ChannelIdType createChannel (
        in ChannelConfigType        configInfo
) raises (AssuranceLevel, CertificateNotRequired, ChanTypeAlgorithmMismatch, DeviceError,
InvalidAlgorithmId, InvalidCertificateId, InvalidKeyId, InvalidMode, InvalidPolicyId,
InvalidProperty, KeyAlgorithmMismatch, KeyExpired, NotCOMSECAlgorithm,
ResourcesUnavailable, UnknownError);
```

4.5.1.2 Parameters.

configInfo

The channelConfigType has the following structure:

```
struct ChannelConfigType {
        ChannelType         type;
        IdType              algorithm;
        IdSequenceType      keys;
        IdType              bypassPolicy;
        IdType              certificate;
        CF::StringSequence  modes;
        CF::Properties      properties;
};
```

type

Identifies the type of crypto channel to create:

CT_SIMPLEX_RX,

Receive only operation.

CT_HALF_DUPLEX,

The channel supports both transmit and receive but only one at a time (the crypto will context switch between receive and transmit portions of algorithm.)

CT_FULL_DUPLEX,

The channel is configured for simultaneous receive and transmit (e.g. not context switching).

CT_BLACK_SIDE,

The channel is configured for black-black encrypt and decrypt (e.g. DAMA order wire).

CT_RED_SIDE

The channel is configured for red-red encrypt and decrypt.

algorithm

Identifies the crypto algorithm to use for the channel.

keys

Identifies the key(s) to use for the channel. Certain waveforms require the use of multiple keys. A key identifier of "" indicates a session key must be generated.

bypassPolicy

Identifies the Bypass Policy to use for the channel. This is a waveform specific bypass policy.

certificate

Only valid for instances where session keys are generated and not pulled from key storage (e.g. Firefly exchange).

modes

The set of modes in which the algorithm will operate.

properties

The set of properties for the algorithm such as straps, seed, etc.

4.5.1.3 State.

N/A.

4.5.1.4 New State.

The state of the new channel is IDLE.

4.5.1.5 Response.

This primitive returns an opaque channel identifier of type ChannelIdType

4.5.1.6 Originator.

This primitive is initiated by the service user.

4.5.1.7 Errors/Exceptions.

The following exceptions may be raised:

AssuranceLevel

The crypto is not certified to operate at the assurance level required by the channel instantiation. Example:

One channel is already running with a SECRET key, the new channel is to be instantiated with a TOP SECRET key and the Crypto is only certified for System High operation.

CertificateNotRequired

A certificate is not required for this channel instantiation.

ChanTypeAlgorithmMismatch

The specified algorithm does not support the requested channel type.

DeviceError

The channel could not be created because of a hardware error.

InvalidAlgorithmId

The algorithm ID is malformed or the algorithm does not exist.

InvalidCertificateId

The certificate ID is malformed or the certificate does not exist.

InvalidKeyId

The key ID is malformed or the key does not exist.

InvalidMode

The mode does not exist.

InvalidPolicyId

The policy ID is malformed or the policy does not exist.

InvalidProperty

The property does not exist.

KeyAlgorithmMismatch

The specified key(s) and algorithm will not work together.

KeyExpired

The specified key(s) have expired and can no longer be used.

NotCOMSECAlgorithm

The specified algorithm is not a COMSEC algorithm but a TRANSEC algorithm.

UnknownError

The channel could not be created because of an indeterminate error.

4.5.2 CRYPT_GET_CHAN_CONFIG.

This primitive retrieves the configuration of an instantiated crypto channel.

4.5.2.1 Synopsis.

```
void getChannelConfig (
        in ChannelIdType            channel,
        out ChannelConfigType       configInfo
) raises (InvalidChannelId);
```

4.5.2.2 Parameters.

channel

The identifier of an instantiated channel.

configInfo

The channel configuration information. See 4.5.1.2 for the definition of ChannelConfigType.

4.5.2.3 State.

The primitive is valid in any state.

4.5.2.4 New State.

The resulting state is unchanged.

4.5.2.5 Response.

N/A.

4.5.2.6 Originator.

This primitive is initiated by the service user.

4.5.2.7 Errors/Exceptions.

The following exceptions may be raised:

InvalidChannelId

The specified channel identifier does not correspond to an instantiated crypto channel.

4.5.3 CRYPT_DESTROY_CHAN.

This primitive destroys an instantiated crypto channel and returns all the resources allocated to it back to the pool of available resources.

4.5.3.1 Synopsis.

```
void destroyChannel (
        in ChannelIdType      channel
) raises (InvalidChannelId, UnknownError);
```

4.5.3.2 Parameters.

channel

Identifies the instantiated channel to destroy.

4.5.3.3 State.

The primitive is valid in any state.

4.5.3.4 New State.

N/A.

4.5.3.5 Response.

N/A.

4.5.3.6 Originator.

This primitive is initiated by the service user.

4.5.3.7 Errors/Exceptions.

The following exceptions may be raised:

InvalidChannelId

The specified channel identifier does not correspond to an instantiated crypto channel.

UnknownError

An error of unidentified origin occurred during channel tear down.

4.5.4 CRYPT_START_CHAN.

This primitive starts a cryptographic channel for an identified mode.

4.5.4.1 Synopsis.

```
void startChannel (
        in ChannelIdType    channel,
        in string           mode
) raises (ChannelAlreadyStarted, ChannelInAlarm, DeviceError, InvalidChannelId, InvalidMode,
UnknownError);
```

4.5.4.2 Parameters.

channel

The identifier of the channel to start

mode

The mode of the channel to start

4.5.4.3 State.

This primitive is valid in the IDLE state.

4.5.4.4 New State.

The resulting state is ACTIVE.

4.5.4.5 Response.

N/A.

4.5.4.6 Originator.

This primitive is initiated by the service user.

4.5.4.7 Errors/Exceptions.

ChannelAlreadyStarted

The channel has already been started.

ChannelInAlarm

The channel is in crypto alarm and cannot be used.

DeviceError

A device error has occurred.

InvalidChannelId

The specified channel identifier does not correspond to an instantiated crypto channel.

InvalidMode

No such mode is available for the instantiated channel.

UnknownError

An unknown error has occurred.

4.5.5 CRYPT_STOP_CHAN.

This primitive stops a channel.

4.5.5.1 Synopsis.

```
void stopChannel (
        in ChannelIdType    channel,
        in string           mode
) raises (ChannelInAlarm, ChannelNotStarted, DeviceError, InvalidChannelId, InvalidMode,
UnknownError);
```

4.5.5.2 Parameters.

channel

Identifies the channel to stop.

mode

Identifies the mode to stop.

4.5.5.3 State.

This primitive is valid in the ACTIVE state.

4.5.5.4 New State.

The resulting state is unchanged.

4.5.5.5 Response.

N/A.

4.5.5.6 Originator.

This primitive is initiated by the service user.

4.5.5.7 Errors/Exceptions.

ChannelInAlarm

The channel is in crypto alarm and cannot be used.

ChannelNotStarted

The channel was never started.

DeviceError

A device error has occurred.

InvalidChannelId

The specified channel identifier does not correspond to an instantiated crypto channel.

InvalidMode

No such mode is available for the instantiated channel.

UnknownError

An unknown error has occurred.

4.5.6 CRYPT_RESET_CHAN.

This primitive resets a crypto channel. All internal states are reset to their default values.

4.5.6.1 Synopsis.

```
void resetChannel (
        in ChannelIdType        channel
) raises (DeviceError, InvalidChannelId, UnknownError);
```

4.5.6.2 Parameters.

channel

Identifies the channel to reset.

4.5.6.3 State.

This primitive is valid in any state.

4.5.6.4 New State.

The resulting state is of the channel is the default state upon channel creation.

4.5.6.5 Response.

N/A.

4.5.6.6 Originator.

This primitive is initiated by the service user.

4.5.6.7 Errors/Exceptions.

The following exceptions may be raised:

DeviceError

A device error occurred on the channel and the channel could not reset properly.

InvalidChannelId

The channel identifier does not correspond to an instantiated channel.

UnknownError

An unknown error occurred on the channel and the channel could not reset properly.

4.5.7 CRYPT_RESET.

This primitive forces a reset of the entire cryptographic subsystem.

4.5.7.1 Synopsis.

void resetCrypto ();

4.5.7.2 Parameters.

N/A.

4.5.7.3 State.

The primitive is valid in any state.

4.5.7.4 New State.

The resulting state is IDLE.

4.5.7.5 Response.

N/A.

4.5.7.6 Originator.

This primitive is initiated by the service user.

4.5.7.7 Errors/Exceptions.

N/A.

4.5.8 CRYPT_ENCRYPT.

This primitive is used for red-red or black-black encryption. It encrypts a sequence of octets and returns the encrypted sequence to the service user.

4.5.8.1 Synopsis.

```
void Encrypt (
        inout CF::OctetSequence     data
) raises (ChannelInAlarm, DeviceError, InvalidChannelId, UnknownError);
```

4.5.8.2 Parameters.

data

Upon entry: The data to encrypt. Upon exit: the encrypted data.

4.5.8.3 State.

The primitive is valid in the ACTIVE state.

4.5.8.4 New State.

The resulting state is unchanged.

4.5.8.5 Response.

N/A.

4.5.8.6 Originator.

This primitive is initiated by the service user.

4.5.8.7 Errors/Exceptions.

ChannelInAlarm

The channel is in crypto alarm and cannot be used.

DeviceError

A device error has occurred.

UnknownError

An error of indeterminate origin occurred.

4.5.9 CRYPT_DECRYPT.

This primitive is used for red-red or black-black decryption. It decrypts a sequence of octets and returns the decrypted sequence to the service user.

4.5.9.1 Synopsis.

```
void Decrypt (
        inout CF::OctetSequence     data
) raises (ChannelInAlarm, DeviceError, UnknownError);
```

4.5.9.2 Parameters.

data

Upon entry: The data to decrypt. Upon exit: the decrypted data.

4.5.9.3 State.

The primitive is valid in the ACTIVE state.

4.5.9.4 New State.

The resulting state is unchanged.

4.5.9.5 Response.

N/A.

4.5.9.6 Originator.

This primitive is initiated by the service user.

4.5.9.7 Errors/Exceptions.

ChannelInAlarm

The channel is in crypto alarm and cannot be used.

DeviceError

A device error has occurred.

UnknownError

An error of indeterminate origin occurred.

4.5.10 CRYPT_ENCRYPT_WITH_ID.

This primitive is used for red-red or black-black encryption. It encrypts a sequence of octets and returns the encrypted sequence to the service user. Multiple channels are multiplexed through the interface by channel ID.

4.5.10.1 Synopsis.

```
void Encrypt (
        in ChannelIdType            channel,
        inout CF::OctetSequence     data
) raises (ChannelInAlarm, DeviceError, InvalidChannelId, UnknownError);
```

4.5.10.2 Parameters.

channel

The identifier for the channel to use for the encryption.

data

The data to encrypt and the returned encrypted data.

4.5.10.3 State.

The primitive is valid in the ACTIVE state.

4.5.10.4 New State.

The resulting state is unchanged.

4.5.10.5 Response.

N/A.

4.5.10.6 Originator.

This primitive is initiated by the service user.

4.5.10.7 Errors/Exceptions.

ChannelInAlarm

The channel is in crypto alarm and cannot be used.

DeviceError

A device error has occurred.

InvalidChannelId

The specified channel identifier does not correspond to an instantiated crypto channel.

UnknownError

An error of indeterminate origin occurred.

4.5.11 CRYPT_DECRYPT_WITH_ID.

This primitive is used for red-red or black-black decryption. It decrypts a sequence of octets and returns the decrypted sequence to the service user. Multiple channels are multiplexed through the interface by channel ID.

4.5.11.1 Synopsis.

```
void Decrypt (
        in ChannelIdType            channel,
        inout CF::OctetSequence     data
) raises (ChannelInAlarm, DeviceError, InvalidChannelId, UnknownError);
```

4.5.11.2 Parameters.

channel

The identifier for the channel to use for the decryption.

data

The data to decrypt and the returned decrypted data.

4.5.11.3 State.

The primitive is valid in the ACTIVE state.

4.5.11.4 New State.

The resulting state is unchanged.

4.5.11.5 Response.

N/A.

4.5.11.6 Originator.

This primitive is initiated by the service user.

4.5.11.7 Errors/Exceptions.

ChannelInAlarm

The channel is in crypto alarm and cannot be used.

DeviceError

A device error has occurred.

InvalidChannelId

The specified channel identifier does not correspond to an instantiated crypto channel.

UnknownError

An error of indeterminate origin occurred.

4.5.12 CRYPT_TRANSFORM_REQ.

This primitive performs red-black encryption and black-red decryption. The location of the object that realizes the interface determines whether decryption or encryption occurs.

4.5.12.1 Synopsis.

```
oneway void transform (
        in any                bypass,
        in CF::OctetSequence payload
);
```

4.5.12.2 Parameters.

bypass

Waveform specific header information to be bypassed through the crypto (e.g. addresses, and real time control).

payload

The payload to be encrypted/decrypted.

4.5.12.3 State.

The primitive is valid in the ACTIVE state.

4.5.12.4 New State.

The resulting state is unchanged.

4.5.12.5 Response.

N/A.

4.5.12.6 Originator.

This primitive is initiated by the service user. Note: this same primitive is initiated by the service provider on the opposite side of the cryptographic boundary after the transform is complete. The service user therefore must implement this interface for waveform data transfer out of the cryptographic subsystem.

4.5.12.7 Errors/Exceptions.

N/A.

4.5.13 CRYPT_TRANSFORM_REQ_WITH_ID.

This primitive performs red-black encryption and black-red decryption. The location of the object that realizes the interface determines whether decryption or encryption occurs. Multiple channels are multiplexed through the interface by channel ID.

4.5.13.1 Synopsis.

```
oneway void transform (
        in ChannelIdType        channel,
        in any                  bypass,
        in CF::OctetSequence    payload
);
```

4.5.13.2 Parameters.

channel

The identifier for the channel to use for the encryption/decryption.

bypass

Waveform specific header information to be bypassed through the crypto (e.g. addresses, and real time control).

payload

The payload to be encrypted/decrypted.

4.5.13.3 State.

The primitive is valid in the ACTIVE state.

4.5.13.4 New State.

The resulting state is unchanged.

4.5.13.5 Response.

N/A.

4.5.13.6 Originator.

This primitive is initiated by the service user.

4.5.13.7 Errors/Exceptions.

N/A.

4.6 KEY.

The Key Management Service is a specialization of the Fill::Manager as Figure 4-8 illustrates.

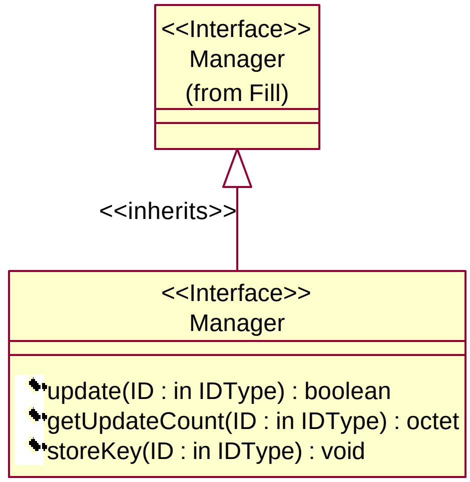

Figure 4-8. Class Diagram: Key Management Service

4.6.1 KEY_ZEROIZE.

This primitive deletes all instances of a single key from a security service as specified by the ID. See paragraph 4.2.9 for the semantics and behavior.

4.6.2 KEY_ZEROIZE_ALL.

This primitive deletes all keys from a security service. See paragraph 4.2.10 for the semantics and behavior.

4.6.3 KEY_GET_IDS.

This primitive retrieves the identifiers of all the keys resident in a security service. See paragraph 4.2.11 for the semantics and behavior.

4.6.4 KEY_EXPIRY.

This primitive retrieves the expiration date and time for a given key within a security service. See paragraph 4.2.12 for the semantics and behavior.

4.6.5 KEY_UPDATE.

This primitive updates a key, which in effect creates a new key. The ID remains the same but the update count is incremented by one. Update counts of a key must remain in sync for peers to communicate.

4.6.5.1 Synopsis.

```
boolean update (
        in IdType       id
) raises (InvalidId, KeyInUse);
```

4.6.5.2 Parameters.

id

The identifier of the key to update.

4.6.5.3 State.

The primitive is valid in any state.

4.6.5.4 New State.

The resulting state is unchanged.

4.6.5.5 Response.

N/A.

4.6.5.6 Originator.

This primitive is initiated by the service user.

4.6.5.7 Errors/Exceptions.

The following exception may be raised:

InvalidId

The key ID is either malformed or the key does not exist.

KeyInUse

The key cannot be update because it key is being used by an instantiated channel.

4.6.6 KEY_GET_UPDATE_COUNT.

This primitive retrieves the current update count for a key.

4.6.6.1 Synopsis.

```
octet getUpdateCount (
        in IdType       id
) raises (InvalidId);
```

4.6.6.2 Parameters.

id

The identifier of the key for which to retrieve the update count.

4.6.6.3 State.

The primitive is valid in any state.

4.6.6.4 New State.

The resulting state is unchanged.

4.6.6.5 Response.

The update count of the key is returned.

4.6.6.6 Originator.

This primitive is initiated by the service user.

4.6.6.7 Errors/Exceptions.

The following exception may be raised:

InvalidId

The key ID is either malformed or the key does not exist.

4.6.7 KEY_STORE_KEY.

This primitive stores a key that is loaded from a DS-102 configured port. Keys loaded by the DS-102 protocol do not have identifiers attached to them. This primitive assigns an identifier.

4.6.7.1 Synopsis.

```
void storeKey (
        in IdType       id
) raises (DuplicateId, InvalidId, NoKey);
```

4.6.7.2 Parameters.

id

The identifier to be associated with the stored key.

4.6.7.3 State.

This primitive is valid in the PENDING_STORE state.

4.6.7.4 New State.

The resulting state is ENABLED.

4.6.7.5 Response.

N/A.

4.6.7.6 Originator.

This primitive is initiated by the service user.

4.6.7.7 Errors/Exceptions.

The following exception may be raised:

DuplicateId

A key already exists with the specified identifier.

InvalidId

The key ID is malformed.

NoKey

There is no key loaded via DS-102 awaiting storage.

4.7 TRANSEC.

Figure 4-9 illustrates the TRANSEC management, control and Key Stream generation services.

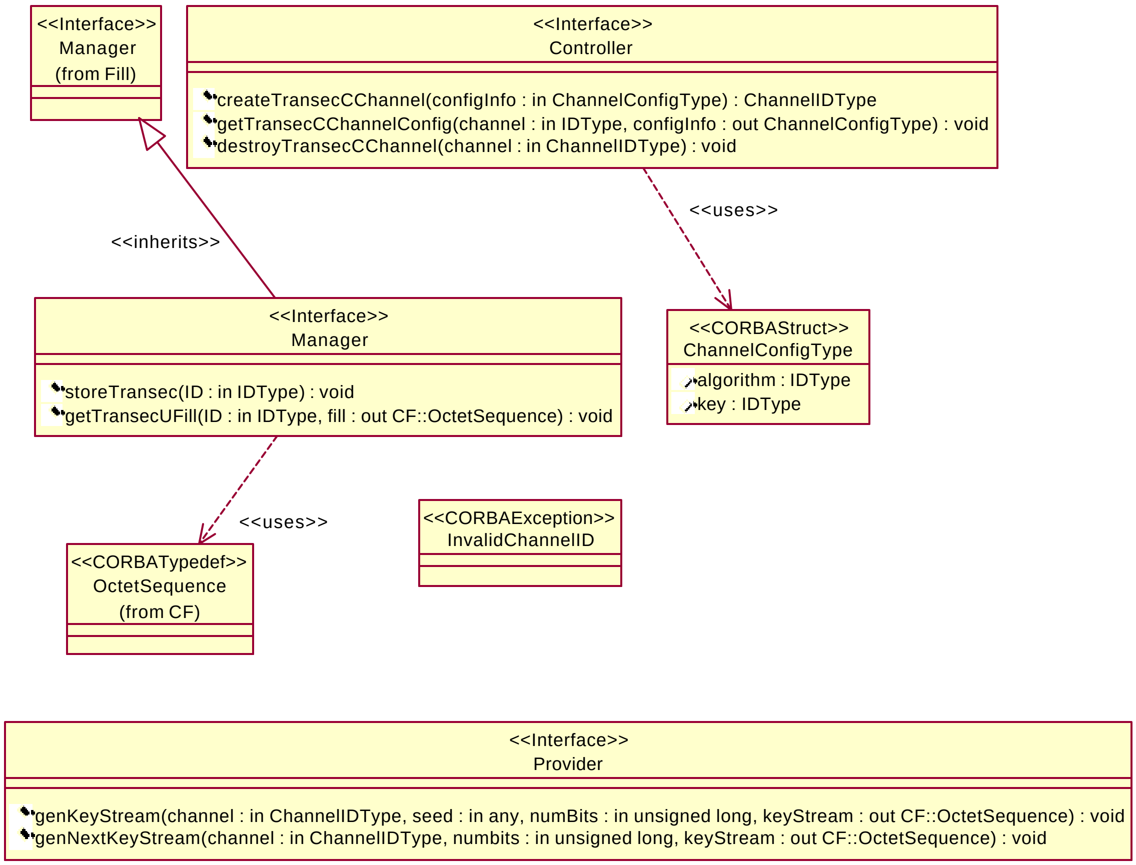

Figure 4-9. Class Diagram: TRANSEC Services

4.7.1 TRAN_CREATE_CHAN.

This primitive creates a classified TRANSEC channel. The channel will generate Key Stream data.

4.7.1.1 Synopsis.

```
ChannelIdType createTransecCChannel (
        in ChannelConfigType        configInfo
) raises (InvalidAlgorithmId, InvalidKeyId, , KeyAlgorithmMismatch, NotTRANSECAlgorithm,
ResourcesUnavailable);
```

4.7.1.2 Parameters.

configInfo

The channelConfigType has the following structure:

```
struct ChannelConfigType {
        IdType algorithm;
        IdType key;
};
```

algorithm

Identifies the TRANSEC algorithm.

key

Identifies the TRANSEC key.

4.7.1.3 State.

N/A.

4.7.1.4 New State.

The resulting state is ACTIVE.

4.7.1.5 Response.

N/A.

4.7.1.6 Originator.

This primitive is initiated by the service user.

4.7.1.7 Errors/Exceptions.

The following exceptions may be raised:

InvalidAlgorithmId

The algorithm identifier is malformed or the algorithm does not exist.

InvalidKeyId

The key identifier is malformed or the key does not exist.

KeyAlgorithmMismatch

The selected key cannot be used with the selected algorithm.

NotTRANSECAlgorithm

The selected algorithm is not a TRANSEC algorithm.

ResourcesUnavailable

The required resources are unavailable for instantiating the channel.

4.7.2 TRAN_GET_CHAN_CONFIG.

This primitive retrieves the configuration of an instantiated TRANSEC channel.

4.7.2.1 Synopsis.

```
void getTransecCChannelConfig (
        in IdType                   channel,
        out ChannelConfigType       configInfo
) raises (InvalidChannelId);
```

4.7.2.2 Parameters.

channel

Identifies the instantiated channel from which to retrieve the configuration information.

configInfo

The returned channel configuration information. See paragraph 4.7.1.2 for the structure.

4.7.2.3 State.

This primitive is valid in any state.

4.7.2.4 New State.

The resulting state is unchanged.

4.7.2.5 Response.

N/A.

4.7.2.6 Originator.

This primitive is initiated by the service user.

4.7.2.7 Errors/Exceptions.

InvalidChannelId

The specified channel identifier does not correspond to an instantiated TRANSEC channel.

4.7.3 TRAN_DESTROY_CHAN.

This primitive destroys an instantiated TRANSEC channel and returns all the resources allocated to it back to the pool of available resources.

4.7.3.1 Synopsis.

```
void destroyTransecCChannel (
        in ChannelIdType      channel
)  raises (InvalidChannelId);
```

4.7.3.2 Parameters.

channel

Identifies the instantiated channel to destroy.

4.7.3.3 State.

This primitive is valid in any state.

4.7.3.4 New State.

N/A.

4.7.3.5 Response.

N/A.

4.7.3.6 Originator.

This primitive is initiated by the service user.

4.7.3.7 Errors/Exceptions.

InvalidChannelId

The specified channel identifier does not correspond to an instantiated TRANSEC channel.

4.7.4 TRAN_GEN_KEY_STREAM.

This primitive generates Key Stream data for a TRANSEC channel. The algorithm is re-seeded.

4.7.4.1 Synopsis.

```
void genKeyStream (
        in ChannelIdType          channel,
        in any                    seed,
        in unsigned long          numBits,
        out CF::OctetSequence     keyStream
)  raises (ChannelInAlarm, DeviceError, InvalidChannelId, UnknownError);
```

4.7.4.2 Parameters.

channel

Identifies the instantiated channel.

seed

Identifies the TRANSEC algorithm seed. They type of seed is algorithm dependent and as such is defined as a CORBA any type.

numBits

The number of bits of Key Stream to generate.

keyStream

The generated Key Stream

4.7.4.3 State.

This primitive is valid in the ACTIVE state.

4.7.4.4 New State.

The resulting state is unchanged.

4.7.4.5 Response.

N/A.

4.7.4.6 Originator.

This primitive is initiated by the service user.

4.7.4.7 Errors/Exceptions.

The following exceptions may be raised:

ChannelInAlarm

The channel is in crypto alarm and cannot be used.

DeviceError

A device error has occurred.

InvalidChannelId

The specified channel identifier does not correspond to an instantiated crypto channel.

InvalidSeedType

The type of the TRANSEC seed does not correspond to that required by the algorithm.

InvalidSeedValue

The value of the TRANSEC seed is not valid (e.g. out of range).

UnknownError

An error of indeterminate origin occurred.

4.7.5 TRAN_GEN_NEXT_KEY_STREAM.

This primitive generates Key Stream data for a TRANSEC channel. The algorithm continues generating data based on the seed last input for TRAN_GEN_KEY_STREAM or the seed input for TRAN_CREATE_CHAN.

4.7.5.1 Synopsis.

```
void genNextKeyStream (
        in ChannelIdType            channel,
        in unsigned long            numbits,
        out CF::OctetSequence       keyStream
) raises (ChannelInAlarm, DeviceError, InvalidChannelId, UnknownError);
```

4.7.5.2 Parameters.

channel

Identifies the instantiated channel.

numBits

The number of bits of Key Stream to generate.

keyStream

The generated Key Stream

4.7.5.3 State.

This primitive is valid in the ACTIVE state.

4.7.5.4 New State.

The resulting state is unchanged.

4.7.5.5 Response.

N/A.

4.7.5.6 Originator.

This primitive is initiated by the service user.

4.7.5.7 Errors/Exceptions.

The following exceptions may be raised:

ChannelInAlarm

The channel is in alarm and cannot be used.

DeviceError

A device error has occurred.

InvalidChannelId

The specified channel identifier does not correspond to an instantiated crypto channel.

UnknownError

An error of indeterminate origin occurred.

4.7.6 TRAN_ZEROIZE.

This primitive deletes all instances of a single TRANSEC load from a security service as specified by the ID. See paragraph 4.2.9 for the semantics and behavior.

4.7.7 TRAN_ZEROIZE_ALL.

This primitive deletes all TRANSEC loads from a security service. See paragraph 4.2.10 for the semantics and behavior.

4.7.8 TRAN_GET_IDS.

This primitive retrieves the identifiers of all the TRANSEC loads resident in a security service. See paragraph 4.2.11 for the semantics and behavior.

4.7.9 TRAN_EXPIRY.

This primitive retrieves the expiration date and time for a given TRANSEC load within a security service. See paragraph 4.2.12 for the semantics and behavior.

4.7.10 TRAN_STORE.

This primitive stores TRANSEC information that is loaded from a DS-102 configured port. TRANSEC information loaded by the DS-102 protocol does not have identifiers attached to it. This primitive assigns an identifier.

4.7.10.1 Synopsis.

```
void storeTransec (
        in IdType id
) raises (DuplicateId, InvalidId);
```

4.7.10.2 Parameters.

id

The identifier to be associated with the stored TRANSEC fill information.

4.7.10.3 State.

This primitive may only be issued in the FILL_STATE_ PENDING_STORE state.

4.7.10.4 New State.

The resulting state is FILL_STATE_ENABLED.

4.7.10.5 Response.

N/A.

4.7.10.6 Originator.

This primitive is initiated by the service user.

4.7.10.7 Errors/Exceptions.

The following exceptions may be raised:

DuplicateId

TRANSEC information already exists with the specified identifier.

InvalidId

The key ID is malformed.

4.7.11 TRAN_GET_FILL.

This primitive retrieves unclassified fill information for use by TRANSEC algorithms that reside outside the cryptographic boundary.

4.7.11.1 Synopsis.

```
void getTransecUFill (
        in IdType                   id,
        out CF::OctetSequence       fill
) raises (InvalidId);
```

4.7.11.2 Parameters.

id

The identifier of the TRANSEC fill to retrieve.

fill

The retrieved unclassified TRANSEC fill information.

4.7.11.3 State.

This primitive can be issued from any state.

4.7.11.4 New State.

The resulting state is unchanged.

4.7.11.5 Response.

N/A.

4.7.11.6 Originator.

This primitive is initiated by the service user.

4.7.11.7 Errors/Exceptions.

The following exception may be raised:

InvalidId

The identifier is malformed or the TRANSEC fill information does not exist.

4.8 POLICY.

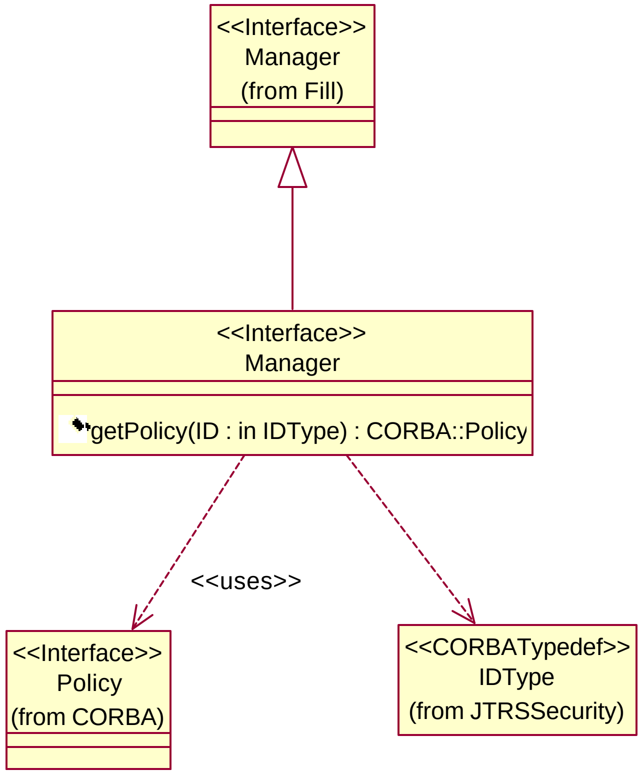

Figure 4-10. Class Diagram: Policy Management Service

4.8.1 POL_ZEROIZE.

This primitive deletes all instances of a single policy from a security service as specified by the ID. See paragraph 4.2.9 for the semantics and behavior.

4.8.2 POL_ZEROIZE_ALL.

This primitive deletes all policies from a security service. See paragraph 4.2.10 for the semantics and behavior.

4.8.3 POL_GET_IDS.

This primitive retrieves the identifiers of all the policies resident in a security service. See paragraph 4.2.11 for the semantics and behavior.

4.8.4 POL_EXPIRY.

This primitive retrieves the expiration date and time for a given policy within a security service. See paragraph 4.2.12 for the semantics and behavior.

4.8.5 POL_GET_POLICY.

This primitive retrieves a policy from the Policy Manager. The information contained in the policy may then be used by an enforcement mechanism to implement access control, bypass filtering, etc. {Note: Policies enter the cryptographic subsystem in XML format. The XML definitions for specific policies have yet to be defined. For policies that are used outside the cryptographic boundary, IDL definitions must exist. The IDL definitions will be specializations of the CORBA::Policy interface. These definitions have yet to be defined as well.}

4.8.5.1 Synopsis.

```
CORBA::Policy getPolicy (
        in IdType       id
) raises (InvalidId);
```

4.8.5.2 Parameters.

id

The identifier of the policy to retrieve

4.8.5.3 State.

N/A.

4.8.5.4 New State.

N/A.

4.8.5.5 Response.

A CORBA Policy is returned. The policy may be narrowed to the specific policy type of interest.

4.8.5.6 Originator.

This primitive is initiated by the service user.

4.8.5.7 Errors/Exceptions.

The following exception may be raised:

InvalidId

The policy identifier is malformed or the policy does not exist.

4.9 INTEGRITY AND AUTHENTICATION.

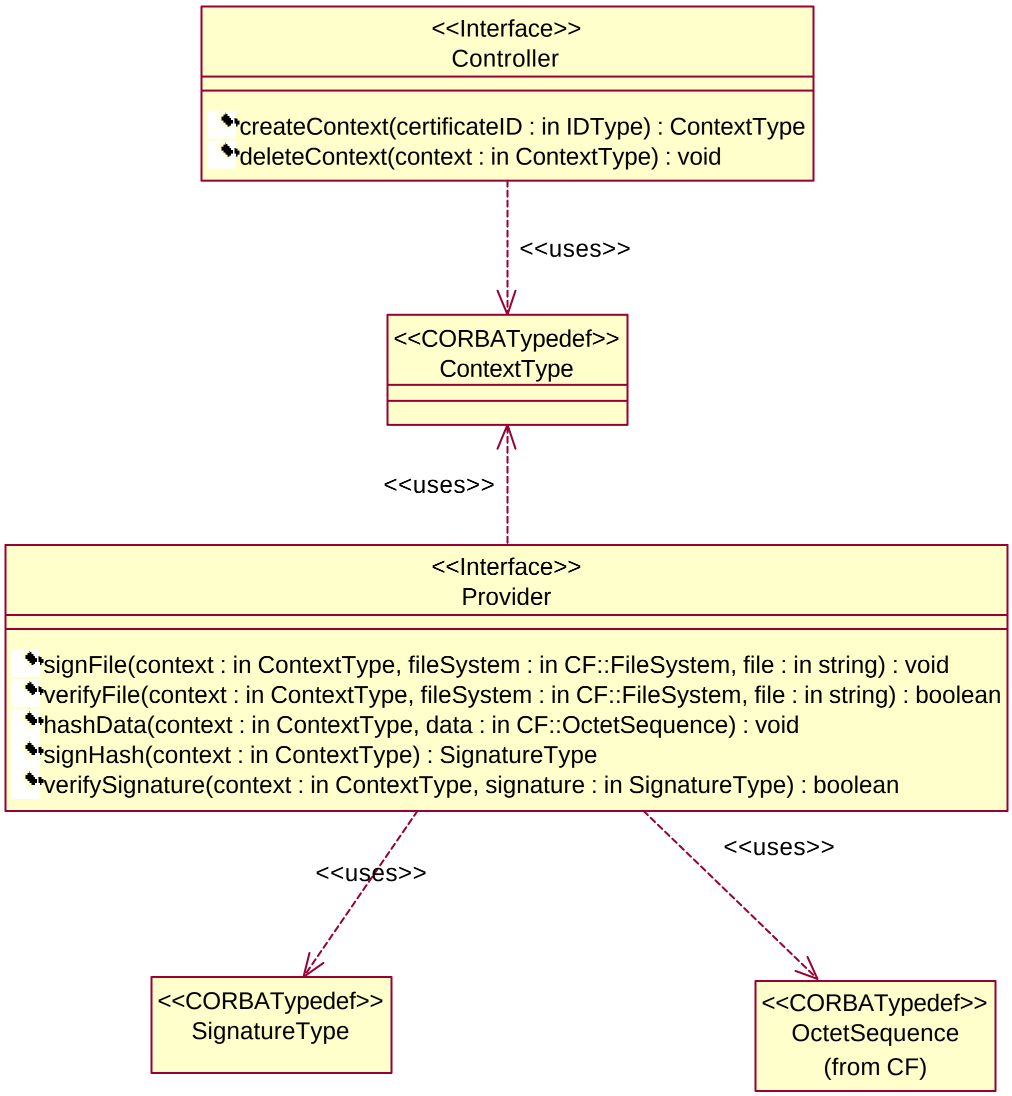

Figure 4-11. Class Diagram: Integrity and Authentication Services

Figure 5-4 shows the states of an Integrity and Authentication context from creation to destruction.

4.9.1 IA_CREATE_CONTEXT.

This primitive creates a context for performing integrity and authentication. The primitive returns a context for use in subsequent invocation of Integrity and Authentication primitives.

4.9.1.1 Synopsis.

```
ContextType createContext (
        in IdType       certificateId
) raises (CertificateExpired, InvalidCertificateId);
```

4.9.1.2 Parameters.

certificateId

The identifier of the certificate to use for the context. The certificate identifies the private and public key pairs to be used.

4.9.1.3 State.

There is no state until a context is created.

4.9.1.4 New State.

The resulting state is HASH_INITIALIZED.

4.9.1.5 Response.

The primitive returns and opaque handle which represents the created context.

4.9.1.6 Originator.

This primitive is initiated by the service user.

4.9.1.7 Errors/Exceptions.

The following exceptions are raised:

CertificateExpired

The identified certificate has expired and cannot be used.

InvalidCertificateId

The certificate identifier is malformed or the certificate does not exist.

4.9.2 IA_DESTROY_CONTEXT .

This primitive destroys the context created by the IA_CREATE_CONTEXT primitive.

4.9.2.1 Synopsis.

```
void deleteContext (
        in ContextType          contextId
) raises (InvalidContextId);
```

4.9.2.2 Parameters.

contextId

The identifier of the context to destroy.

4.9.2.3 State.

This primitive may issued from any state.

4.9.2.4 New State.

There is no state as the context is destroyed.

4.9.2.5 Response.

N/A.

4.9.2.6 Originator.

This primitive is initiated by the service user.

4.9.2.7 Errors/Exceptions.

The following exception is raised:

InvalidContextId

The context identifier does not correspond to a valid context.

4.9.3 IA_SIGN_FILE.

This primitive performs a secure hash of the contents of a file, signs the hash, attaches the signature to the file and initializes the internal hash associated with the context. This primitive may be called multiple times with the same context to sign multiple files.

4.9.3.1 Synopsis.

```
void signFile (
        in ContextType        contextId,
        in CF::FileSystem     fileSystem,
        in string             file
) raises (InvalidContextId, HashNotInitialized);
```

4.9.3.2 Parameters.

contextId

The identifier of the context to use for signing the file.

fileSystem

Identifies the location of the file to be signed.

file

The name of the file to be signed.

4.9.3.3 State.

This primitive may only be issued when the context is in the HASH_INITIALIZED state.

4.9.3.4 New State.

The resulting state is unchanged.

4.9.3.5 Response.

N/A.

4.9.3.6 Originator.

This primitive is initiated by the service user.

4.9.3.7 Errors/Exceptions.

InvalidContextId

The context identifier does not correspond to a valid context.

HashNotInitialized

The context's current state is HASH_IN_PROGRESS and cannot be used for signing or verifying whole files until the current hash has been signed or verified.

4.9.4 IA_VERIFY_FILE.

This primitive decrypts the signature attached to the file into a hash value, performs a secure hash of the contents of the file and compares the decrypted hash with the computed hash. If they are identical, the file is verified. The internal hash is initialized upon completion. This primitive may be called multiple times with the same context to verify multiple files.

4.9.4.1 Synopsis.

```
boolean verifyFile (
        in ContextType          contextId,
        in CF::FileSystem       fileSystem,
        in string               file
) raises (InvalidContextId, HasNotInitialized);
```

4.9.4.2 Parameters.

contextId

Identifies the context to use to verify the file.

fileSystem

Identifies the location of the file to be verified.

file

The name of the file to be verified.

4.9.4.3 State.

This primitive may only be issued when the context is in the HASH_INITIALIZED state.

4.9.4.4 New State.

The resulting state is unchanged.

4.9.4.5 Response.

This primitive returns a boolean:

FALSE The file is not verified.

TRUE The file is verified.

4.9.4.6 Originator.

This primitive is initiated by the service user.

4.9.4.7 Errors/Exceptions.

InvalidContextId

The context identifier does not correspond to a valid context.

HashNotInitialized

The context's current state is HASH_IN_PROGRESS and cannot be used for signing or verifying whole files until the current hash has been signed or verified.

4.9.5 IA_HASH.

This primitive performs a secure hash of a block of data. This function may be called iteratively to compute the hash over several blocks of data. The IA_SIGN_FILE or IA_VERIFY_FILE primitives may not be called once this primitive is invoked for a context until an IA_SIGN_HASH or IA_VERIFY primitive has been called.

4.9.5.1 Synopsis.

```
void hashData (
        in ContextType          contextId,
        in CF::OctetSequence data
) raises (InvalidContextId);
```

4.9.5.2 Parameters.

contextId

Identifies the context to use to verify the file.

data

The block of data for which to compute the hash.

4.9.5.3 State.

This primitive may be issued in any state.

4.9.5.4 New State.

The resulting state is HASH_IN_PROGRESS.

4.9.5.5 Response.

N/A.

4.9.5.6 Originator.

This primitive is initiated by the service user.

4.9.5.7 Errors/Exceptions.

InvalidContextId

The context identifier does not correspond to a valid context.

4.9.6 IA_SIGN_HASH.

This primitive signs the computed hash value held within the integrity and authentication context and returns the digital signature to the caller. The internal hash is initialized upon completion.

4.9.6.1 Synopsis.

```
SignatureType signHash (
        in ContextType          contextId
) raises (InvalidContextId);
```

4.9.6.2 Parameters.

contextId

Identifies the context to use sign the hash.

4.9.6.3 State.

This primitive may only be issued in the HASH_IN_PROGRESS state.

4.9.6.4 New State.

The resulting state is HASH_INITIALIZED.

4.9.6.5 Response.

N/A.

4.9.6.6 Originator.

This primitive is initiated by the service user.

4.9.6.7 Errors/Exceptions.

InvalidContextId

The context identifier does not correspond to a valid context.

NoHashCalculated

The context's current state is HASH_INITIALZED. There have been no invocations of the IA_HASH primitive to calculate a hash value.

4.9.7 IA_VERIFY_HASH.

This primitive decrypts the provided signature and compares it with the computed hash value held within the integrity and authentication context. The internal hash is initialized upon completion.

4.9.7.1 Synopsis.

```
boolean verifySignature (
        in ContextType          contextId,
        in SignatureType        signature
) raises (InvalidContextId);
```

4.9.7.2 Parameters.

contextId

Identifies the context to use sign the verify the hash.

signature

The signature to decrypt and compare against the computed hash.

4.9.7.3 State.

This primitive may only be issued in the HASH_IN_PROGRESS state.

4.9.7.4 New State.

The resulting state is HASH_INITIALIZED.

4.9.7.5 Response.

This primitive returns a boolean:

FALSE The hash is not verified.

TRUE The hash is verified.

4.9.7.6 Originator.

This primitive is initiated by the service user.

4.9.7.7 Errors/Exceptions.

InvalidContextId

The context identifier does not correspond to a valid context.

NoHashCalculated

The context's current state is HASH_INITIALZED. There have been no invocations of the IA_HASH primitive to calculate a hash value.

4.10 ALARM.

Figure 4-12 shows the type definitions defined for audit records to be logged when security alarms occur. {Note: It is assumed that the CF::Logger interface will be used for this function. The CF::Logger as it is defined currently does not support this but a change proposal is pending to make the necessary changes.} The alarm record covers many different types of alarms and is not available for general consumption. Access to the records will be limited by security policies.

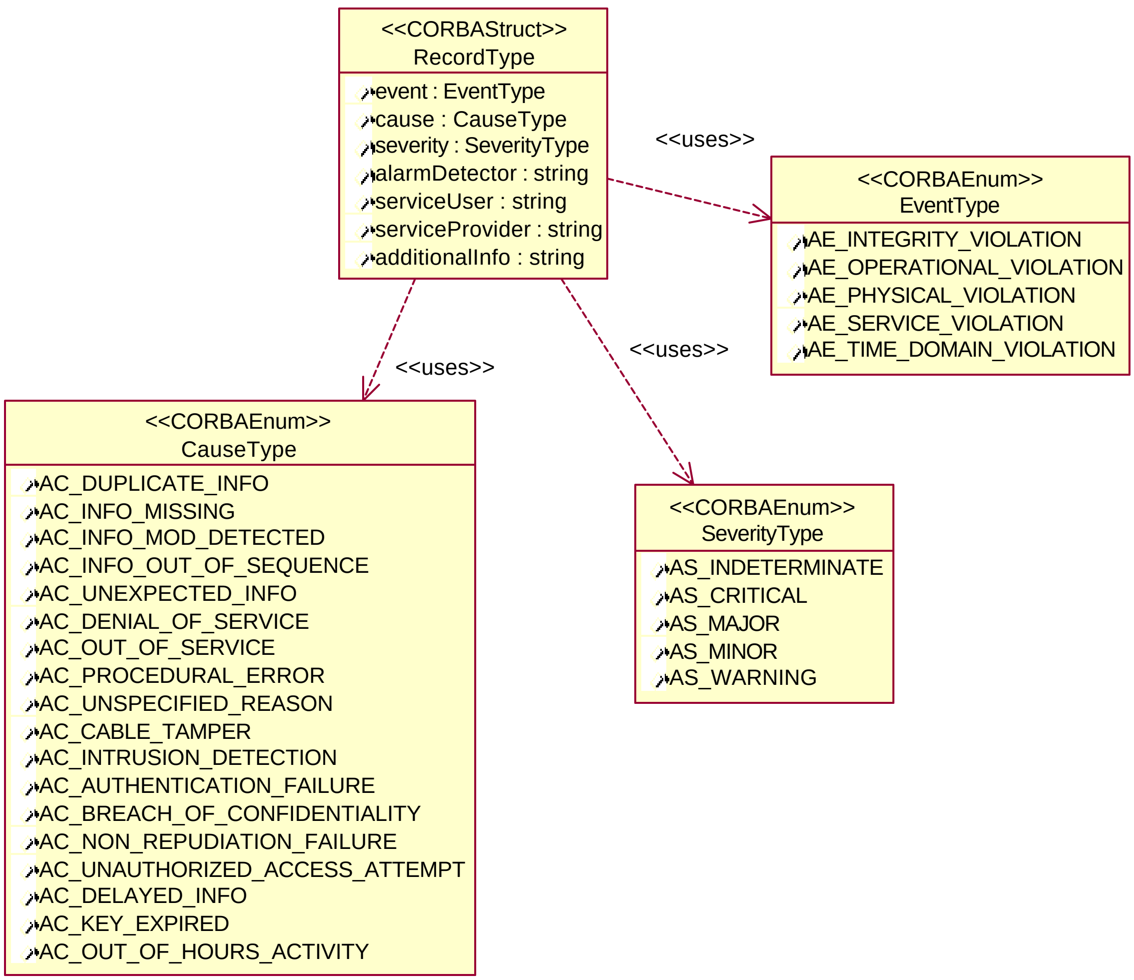

Figure 4-12. Class Diagram: Alarm Type Definitions

Figure 4-13 shows the alarm service it self. This is a service for notifying the Security Service User and is a simple indicator that a Crypto Alarm has occurred. There are no user access controls on this indicator. The implementation of the service resides with Security Service User. The Security Service merely invokes it when a crypto Alarm has occurred.

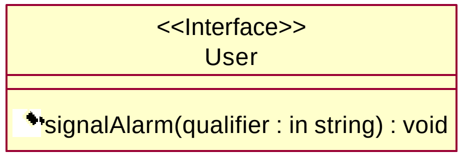

Figure 4-13. Class Diagram: Alarm Service

4.10.1 ALARM_SIGNAL.

This primitive signals a security service user of a crypto alarm.

4.10.1.1 Synopsis.

```
void signalAlarm (
        in string        qualifier
);
```

4.10.1.2 Parameters.

qualifier

A string to provide additional information about the alarm such as a channel identifier.

4.10.1.3 State.

This primitive may only be issued when a Crypto or TRANSEC channel enters the ALARM state.

4.10.1.4 New State.

The state remains unchanged.

4.10.1.5 Response.

N/A.

4.10.1.6 Originator.

This primitive is initiated by the service provider.

4.10.1.7 Errors/Exceptions.

N/A.

4.11 TIME.

The Time Management service is shown in Figure 4-14.

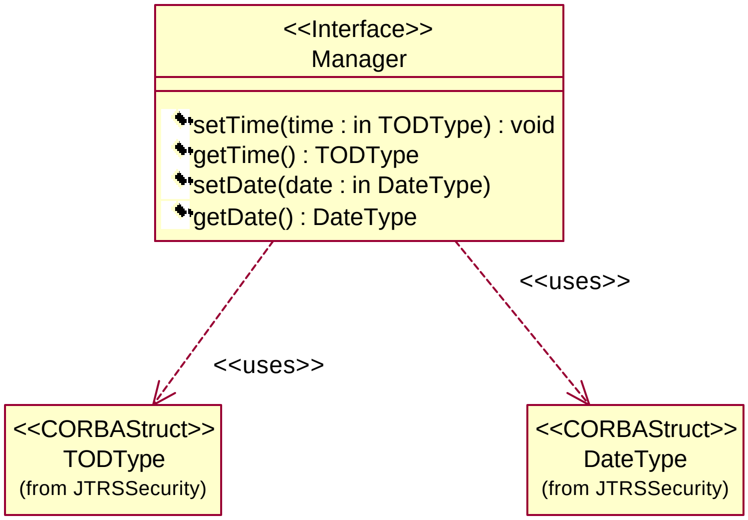

Figure 4-14. Class Diagram: Time Management Service

4.11.1 TIME_SET_TOD

This primitive sets the time of day kept within the cryptographic subsystem.

4.11.1.1 Synopsis.

```
void setTime (
        in TODType   time
) raises (InvalidValue);
```

4.11.1.2 Parameters.

time

The time of day to set.

TODType has the following structure:

```
struct TODType {
        unsigned long seconds;
        unsigned long nanoseconds;
};
```

seconds

The number of seconds past midnight. A value greater than 86399 is invalid and will raise an InvalidValue exception

nanoseconds

The number of nanoseconds since the last increment of *seconds.* A value greater than 999,999,999 is invalid and will raise an InvalidValue exception.

4.11.1.3 State.

N/A.

4.11.1.4 New State.

N/A.

4.11.1.5 Response.

N/A.

4.11.1.6 Originator.

This primitive is initiated by the service user.

4.11.1.7 Errors/Exceptions.

The following exception may be raised:

InvalidValue

The time is not a valid time of day.

4.11.2 TIME_GET_TOD.

This primitive returns the time of day maintained by the cryptographic subsystem.

4.11.2.1 Synopsis.

TODType getTime ();

4.11.2.2 Parameters.

N/A.

4.11.2.3 State.

N/A.

4.11.2.4 New State.

N/A.

4.11.2.5 Response.

The time of day is returned. See paragraph 4.11.1.2 for the structure.

4.11.2.6 Originator.

This primitive is initiated by the service user.

4.11.2.7 Errors/Exceptions.

N/A.

4.11.3 TIME_SET_DATE.

This primitive sets the date kept within the cryptographic subsystem.

4.11.3.1 Synopsis.

```
void setDate (
        in DateType    date
) raises (InvalidValue);
```

4.11.3.2 Parameters.

date

The date to set. DateType has the following structure:

```
struct DateType {
        YearType       year;
        DayType        day;
};
```

year

The year relative to an established reference.

day

The day. Valid values are 1-365 or 366 depending on the year. A value outside this range causes InvalidValue exception to be raised.

4.11.3.3 State.

N/A.

4.11.3.4 New State.

N/A.

4.11.3.5 Response.

N/A.

4.11.3.6 Originator.

This primitive is initiated by the service user.

4.11.3.7 Errors/Exceptions.

The following exception may be raised:

InvalidValue

The date is not a valid date.

4.11.4 TIME_GET_DATE.

This primitive returns the date maintained by the cryptographic subsystem.

4.11.4.1 Synopsis.

DateType getDate ();

4.11.4.2 Parameters.

N/A.

4.11.4.3 State.

N/A.

4.11.4.4 New State.

N/A.

4.11.4.5 Response.

The date is returned. See paragraph 4.11.3.2 for the structure.

4.11.4.6 Originator.

This primitive is initiated by the service user.

4.11.4.7 Errors/Exceptions.

N/A.

4.12 GPS.

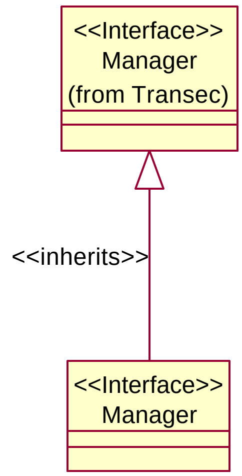

Figure 4-15. Class Diagram: GPS Management Service

4.12.1 GPS_ZEROIZE.

This primitive deletes all instances of a single TRANSEC load from a security service as specified by the ID. See paragraph 4.2.9 for the semantics and behavior.

4.12.2 GPS_ZEROIZE_ALL.

This primitive deletes all TRANSEC loads from a security service. See paragraph 4.2.10 for the semantics and behavior.

4.12.3 GPS_GET_IDS.

This primitive retrieves the identifiers of all the TRANSEC loads resident in a security service. See paragraph 4.2.11 for the semantics and behavior.

4.12.4 GPS_EXPIRY.

This primitive retrieves the expiration date and time for a given TRANSEC load within a security service. See paragraph 4.2.12 for the semantics and behavior.

5 ALLOWABLE SEQUENCE OF SERVICE PRIMITIVES.

There are services within the security API which when implemented require maintenance of state information. This section identifies the states associated with these services and the order in which the service primitives may be invoked.

5.1 FILL STATES.

Table 5-1 describes the states associated with an instantiated crypto channel

Table 5-1. Fill States

STATE	DESCRIPTION
DISABLED	The fill port is disabled
ENABLED	The fill port is enabled
LOAD_IN_PROGRESS	The load of fill information is in progress
PENDING_STORE	The fill information requires an ID to be assigned to it (DS-102 only)

The state diagram in Figure 5-1 illustrates the allowable sequence of primitives for fill operations. DS-101 and DS-102 type fills are differentiated by the need to assign identifiers to DS-102 fill information. The PENDING_STORE state reflects this difference. From a state perspective DS-101 and RS-232 fills can be considered identical.

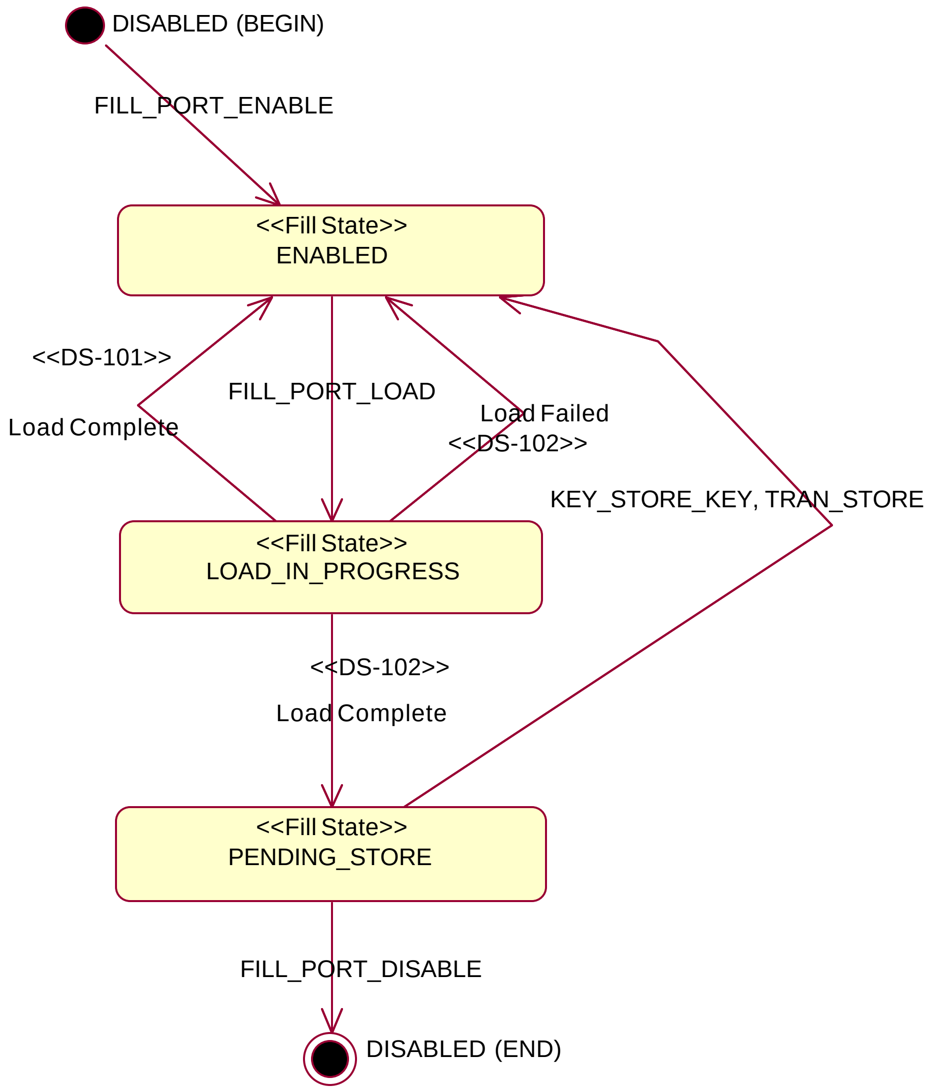

Figure 5-1. Fill State Transitions

5.2 CRYPTO CHANNEL STATES.

Table 5-2 describes the states associated with an instantiated crypto channel.

Table 5-2. Crypto Channel States

STATE	DESCRIPTION
IDLE	Crypto channel is created but has not been started
ACTIVE	Crypto channel has been started in a given mode
ALARM	An alarm has occurred and the channel has been disabled.

The state diagram in Figure 5-2 illustrates the allowable sequence of primitives for a crypto channel from creation to destruction.

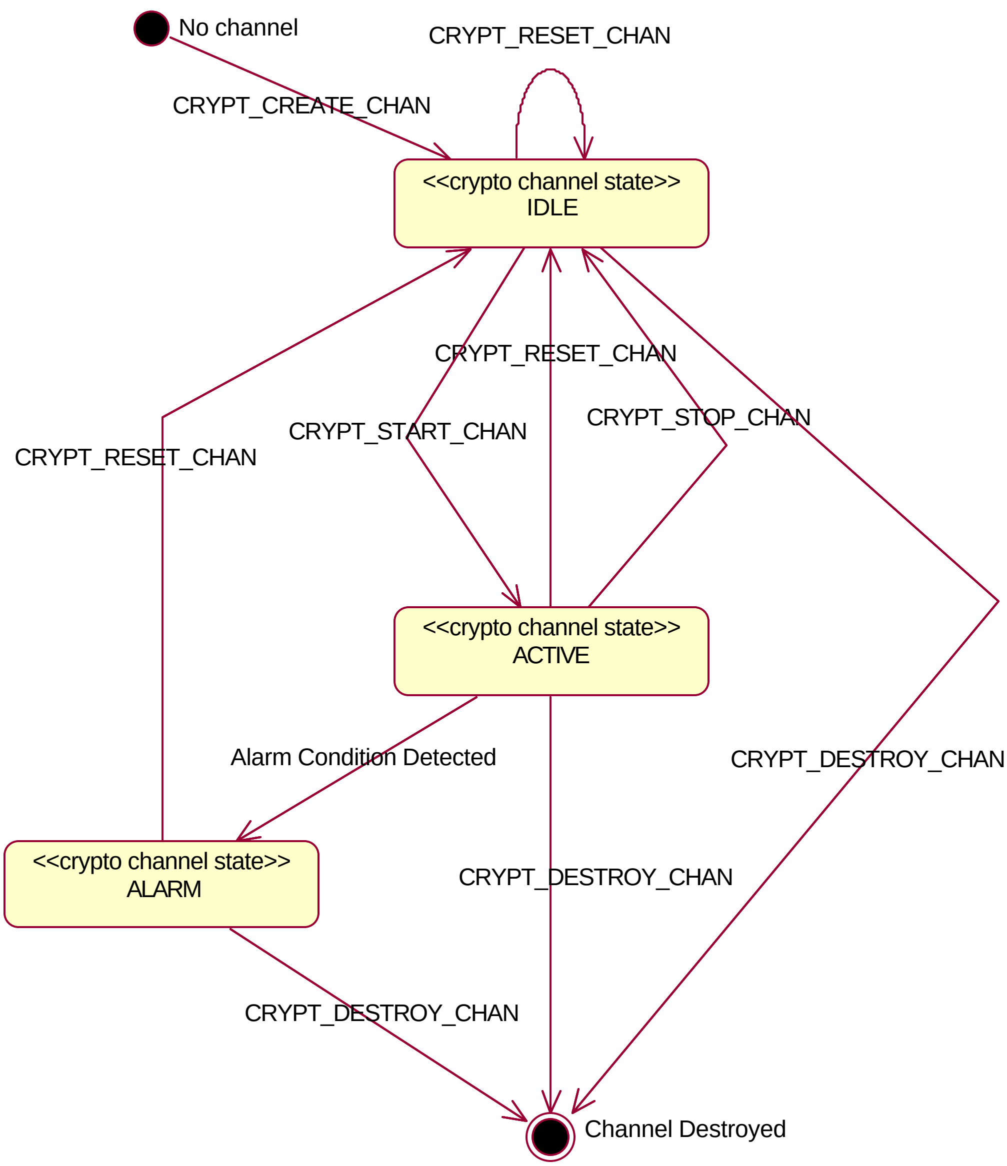

Figure 5-2. State Diagram: Crypto Channel State Transitions

5.3 TRANSEC CHANNEL STATES.

Table 5-3 describes the states associated with an instantiated crypto channel.

Table 5-3. TRANSEC Channel States

STATE	DESCRIPTION
ACTIVE	TRANSEC channel has been started.
ALARM	An alarm has occurred and the channel has been disabled.

The state diagram in Table 5-3 illustrates the allowable sequence of primitives for a crypto channel from creation to destruction.

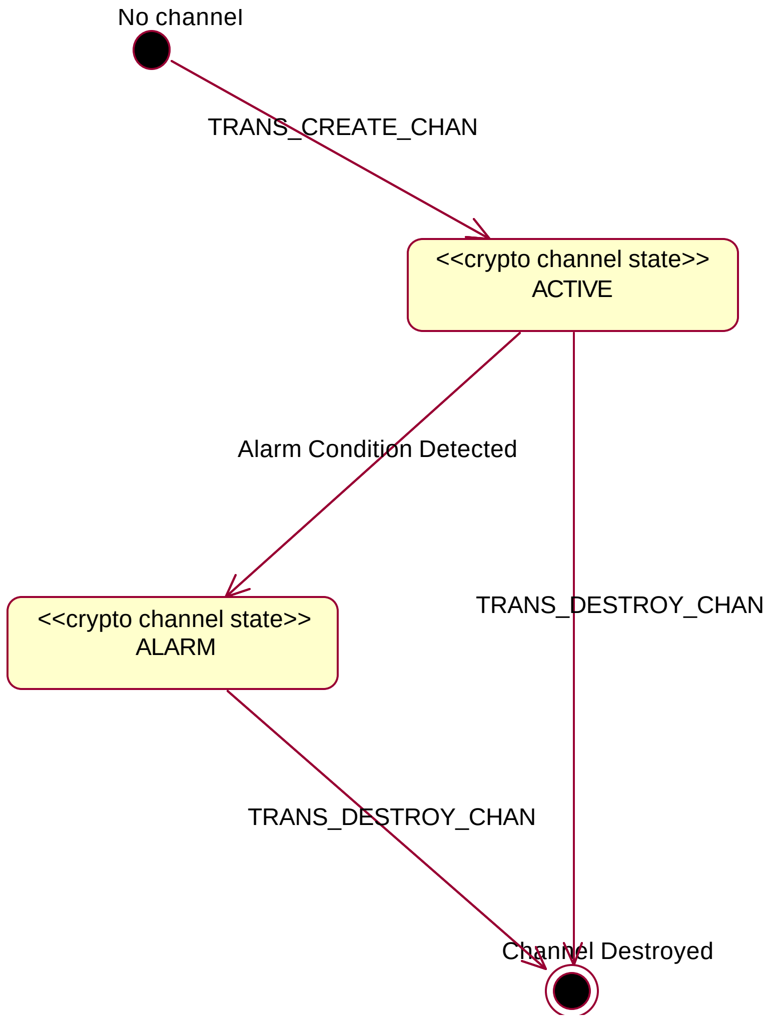

Figure 5-3. State Diagram: TRANSEC Channel State Transitions

5.4 INTEGRITY AND AUTHENTICATION STATES.

Table 5-4 describes the states associated with an Integrity and Authentication context.

Table 5-4. Integrity and Authentication States

STATE	DESCRIPTION
HASH_INITIALIZED	The internal hash associated with the context has been initialized and is ready for use.
HASH_IN_PROGRESS	The hash has been updated from a chunk of data and cannot be used for verification or signature.

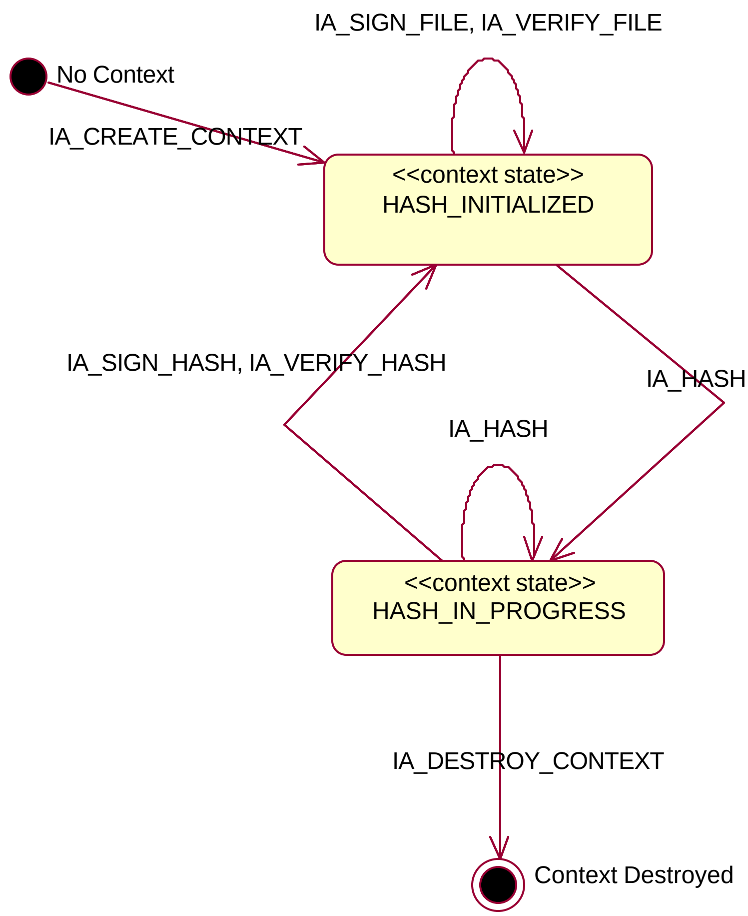

Figure 5-4. State Diagram: Integrity and Authentication Context State Transitions

APPENDIX A. PRECEDENCE OF SERVICE PRIMITIVES

Not applicable.

APPENDIX B. SERVICE USER GUIDELINES

{TBD}.

APPENDIX C. SERVICE PROVIDER-SPECIFIC INFORMATION

{TBD}.

APPENDIX D. IDL

```
#ifndef __JTRSSECURITY_DEFINED
#define __JTRSSECURITY_DEFINED

/* CmIdentification
  %X% %Q% %Z% %W% */

#include "cf.idl"
#include "orb.idl"


module JTRSSecurity {

	typedef string IdType;

	/* Sequence of IDs. Used identify multiple keys, algorithms, etc. */

	typedef sequence <IdType> IdSequenceType;

	/* An example Security Manager that aggregates the fill managers. */

	interface Manager {
		exception ZeroizeFailed {
		};

		/* Zeroize all fill data (keys, algorithms, transec, certificates
and policies).
		@roseuid 39E4D4BD0164 */
		void zeroizeAll ();

	};

	typedef unsigned short YearType;

	typedef octet MonthType;

	typedef octet DayType;

	/* Place holder for date definition. */

	struct DateType {
		YearType year;
		DayType day;
	};

	/* Place holder for TOD definition.  This current definition represents
time past midnight. */

	struct TODType {
		unsigned long seconds;
		unsigned long nanoseconds;
	};
```

```
      /* Used to identify an instantiated crypto or transec channel. */

      typedef unsigned long ChannelIdType;

      module IandA {

            typedef CF::OctetSequence SignatureType;

            typedef unsigned long ContextType;

            exception InvalidContextId {
            };

            /* This interface provides operations to verify the integrity and
authenticity of files and data. A channel of CT_BUSS_FILL must be create
first. */

            interface Provider {
                  exception HashNotInitialized {
                  };

                  exception NoHashCalculated {
                  };

                  /* This operation attaches a digital signature to a file.
                  @roseuid 39EB2CA401C7 */
                  void signFile (
                        in ContextType contextId,
                        in CF::FileSystem fileSystem,
                        in string file
                        )
                        raises (InvalidContextId, HashNotInitialized);

                  /* This operation verifies the digital signature attached
to a file.
                  @roseuid 39EB2CAA0017 */
                  boolean verifyFile (
                        in ContextType contextId,
                        in CF::FileSystem fileSystem,
                        in string file
                        )
                        raises (InvalidContextId, HashNotInitialized);

                  /* This operation hashes the input data into the existing
hash represented by the channel
                  @roseuid 39F0D1E70005 */
                  void hashData (
                        in ContextType contextId,
                        in CF::OctetSequence data
                        )
                        raises (InvalidContextId);

                  /* This operation signs the hash represented by the
channel.
                  @roseuid 39F0CF5C0171 */
```

```
                  SignatureType signHash (
                        in ContextType contextId
                        )
                        raises (InvalidContextId, NoHashCalculated);

                  /* This operation verifies that the input signature matches
the signature generated from the hash represented by the channel
                  @roseuid 39F0CF880142 */
                  boolean verifySignature (
                        in ContextType contextId,
                        in SignatureType signature
                        )
                        raises (InvalidContextId, NoHashCalculated);

            };

            interface Controller {
                  exception InvalidCertificateId {
                  };

                  exception CertificateExpired {
                  };

                  /*
                  @roseuid 3A0452EC01F7 */
                  ContextType createContext (
                        in IdType certificateId
                        )
                        raises (InvalidCertificateId, CertificateExpired);

                  /*
                  @roseuid 3A0452FF015E */
                  void deleteContext (
                        in ContextType contextId
                        )
                        raises (InvalidContextId);

            };

      };

      module Fill {

            /* This enum defines the possible configurations for a fill port.
The load operation will behave differently based on the port configuration.
*/

            enum PortType {
                  PT_DS101,
                  PT_DS102,
                  PT_RS232
            };

            /* This interface must be implemented by the user of a fill port
to support DS102 type fills. */
```

```
            interface PortUser {
                  /* This operation signals the user to connect the fill
device to the fill port.
                  @roseuid 39DB3985034B */
                  void signalConnectDevice (
                        in string instruction
                        );

                  /* This operation signals the user to set the selector on
the DS102 fill device and then invoke the Fill::Port::load operation
                  @roseuid 39DB3869005D */
                  void signalLoad (
                        in string instruction
                        );

                  /* This operation signals the user to assign an ID to the
fill data after being input using the Fill::Port::load operation.
The user will then invoke the requisite storeDS102 operation.
                  @roseuid 39DB2F4D0353 */
                  void signalAssignId (
                        in string instruction
                        );

            };

            /* This is the interface for filling the radio from a file (e.g.
black fills).  A channel of type CT_BUSS_FILL must be created first. */

            interface Bus {
                  exception FileNotValid {
                  };

                  /* This operation loads fill data from a file.
                  @roseuid 39EC9907025F */
                  void load (
                        in CF::FileSystem fileSys,
                        in string fileName
                        );

            };

            enum LoadResultType {
                  LR_COMPLETED,
                  LR_DEVICE_ERROR,
                  LR_CORRUPTED_LOAD
            };

            /* This interface provides functionality for controlling a fill
port. */

            interface Port {
                  /* This operation configures the port for one of
DS101,DS102 or RS232 operation.
                  @roseuid 39DB3B690134 */
                  void configure (
                        in PortType type
```

```
                        );

                  /* This operation enables the fill port represented by an
object with the Fill::Port interface
                  @roseuid 39DB3C03037A */
                  void enable ();

                  /* This operation disables the fill port represented by an
object with the Fill::Port interface
                  @roseuid 39DB3C080273 */
                  void disable ();

                  /* This operation causes data to be loaded from the fill
device into the Fill::Port.
 If the port is configured for DS101 then the load is automated and the fill
information is
automatically distributed to various fill locations.  If the port is
configured for DS102 then only one
fill is performed (i.e. one key, one hopset, etc.).  If the port is
configured for RS-232...
                  @roseuid 39E355E500E9 */
                  LoadResultType load ();

            };

            /* This interface provides for zeroizing and obtaining the
identity of fill data in the radio. */

            interface Manager {
                  exception InvalidId {
                  };

                  exception ZeroizeFailed {
                  };

                  exception ElementInUse {
                  };

                  /* This operation zeroizes the fill element identified by
ID.
                  @roseuid 39E338460149 */
                  void zeroize (
                        in IdType id,
                        in boolean override
                        )
                        raises (InvalidId, ElementInUse, ZeroizeFailed);

                  /* This operation zeroizes all fill elements associated
with the manager. (e.g. keys for a Key Manager)
                  @roseuid 39E33846015D */
                  void zeroizeAll ()
                        raises (ZeroizeFailed);

                  /* This operation gets a list of all the IDs for which the
manager is responsible (e.g. the IDs of all the algorithms loaded in to an
Algorithm Manager.
```

```
                @roseuid 39E338460171 */
                void getIds (
                      out IdSequenceType ids
                      );

                /*
                @roseuid 39EF54900101 */
                boolean expiry (
                      in IdType id,
                      out DateType date,
                      out TODType time
                      )
                      raises (InvalidId);

          };

    };

    module Key {

          /* This interface represents the fill management interface for
key fills. */

          interface Manager : Fill::Manager {
                exception NoKey {
                };

                exception KeyInUse {
                };

                exception DuplicateId {
                };

                /* Store the 102 fill data with the name provided in ID
                @roseuid 39E3637B0072 */
                void storeKey (
                      in IdType id
                      )
                      raises (DuplicateId, InvalidId, NoKey);

                /* Perform a key update on the key identified by ID.
                @roseuid 39DC8745038C */
                boolean update (
                      in IdType id
                      )
                      raises (InvalidId, KeyInUse);

                /* Get the current update count for the key identified by
ID.
                @roseuid 39E359C8000D */
                octet getUpdateCount (
                      in IdType id
                      )
                      raises (InvalidId);

          };
```

```
      };

      module Algorithm {

            /* This interface represents the fill management interface for
algorithm fills. */

            interface Manager : Fill::Manager {
            };

      };

      module Transec {

            /* This structure defines the channel configuration parameters
for a type 1 transec channel. */

            struct ChannelConfigType {
                  /* Identifies the transec algorithm. */
                  IdType algorithm;
                  /* Identifies the transec key. */
                  IdType key;
            };

            exception InvalidChannelId {
            };

            exception InvalidSeedType {
            };

            exception InvalidSeedValue {
            };

            /* This interface represents the fill management interface for
transec fills. */

            interface Manager : Fill::Manager {
                  exception DuplicateId {
                  };

                  /* Store the 102 fill data with the name provided in ID.
                  @roseuid 39E3641B0194 */
                  void storeTransec (
                        in IdType id
                        )
                        raises (DuplicateId, InvalidId);

                  /* Get the type 2 transec fill data identified by ID.
                  @roseuid 39E35DDB01F8 */
                  void getTransecUFill (
                        in IdType id,
                        out CF::OctetSequence fill
                        )
                        raises (InvalidId);
```

```
            };

            /* This interface is used for creating and destroying type 1
transec channels. */

            interface Controller {
                  exception InvalidAlgorithmId {
                  };

                  exception InvalidKeyId {
                  };

                  exception NotTRANSECAlgorithm {
                  };

                  exception ResourcesUnavailable {
                  };

                  exception KeyAlgorithmMismatch {
                  };

                  /* This operation instantiates a type 1 transec channel.
                  @roseuid 39E72FC4008C */
                  ChannelIdType createTransecCChannel (
                        in ChannelConfigType configInfo
                        )
                        raises
(InvalidAlgorithmId,InvalidKeyId,KeyAlgorithmMismatch,NotTRANSECAlgorithm,Res
ourcesUnavailable);

                  /* This operation gets the configuration of a type 1
transec channel.
                  @roseuid 39E735ED010C */
                  void getTransecCChannelConfig (
                        in IdType channel,
                        out ChannelConfigType configInfo
                        )
                        raises (InvalidChannelId);

                  /* This operation destroys a type one transec channel.
                  @roseuid 39F0AC530002 */
                  void destroyTransecCChannel (
                        in ChannelIdType channel
                        )
                        raises (InvalidChannelId);

            };

            /* This interface is used for generating type 1 transec key
streams. */

            interface Provider {
                  exception ChannelInAlarm {
                  };

                  exception DeviceError {
```

```
                };

                exception UnknownError {
                };

                /* Generate a type 1 transec key stream with a new seed.
                @roseuid 39E73749006C */
                void genKeyStream (
                      in ChannelIdType channel,
                      in any seed,
                      in unsigned long numBits,
                      out CF::OctetSequence keyStream
                      )
                      raises (ChannelInAlarm, DeviceError,
InvalidChannelId, InvalidSeedType, InvalidSeedValue, UnknownError);

                /* Generate a type 1 transec key stream without reseeding
the algorithm.
                @roseuid 39E73849034F */
                void genNextKeyStream (
                      in ChannelIdType channel,
                      in unsigned long numbits,
                      out CF::OctetSequence keyStream
                      )
                      raises (ChannelInAlarm, DeviceError,
InvalidChannelId, UnknownError);

          };

    };

    module Alarm {

          /* This enum defines the type of security alarm that will be
generated as an audit event in the logger. */

          enum EventType {
                AE_INTEGRITY_VIOLATION,
                AE_OPERATIONAL_VIOLATION,
                AE_PHYSICAL_VIOLATION,
                AE_SERVICE_VIOLATION,
                AE_TIME_DOMAIN_VIOLATION
          };

          /* This enum defines the severity of the alarm event. */

          enum SeverityType {
                AS_INDETERMINATE,
                AS_CRITICAL,
                AS_MAJOR,
                AS_MINOR,
                AS_WARNING
          };

          /* This is enum indicates the cause of the crypto alarm. */
```

```
            enum CauseType {
                  AC_DUPLICATE_INFO,
                  AC_INFO_MISSING,
                  AC_INFO_MOD_DETECTED,
                  AC_INFO_OUT_OF_SEQUENCE,
                  AC_UNEXPECTED_INFO,
                  AC_DENIAL_OF_SERVICE,
                  AC_OUT_OF_SERVICE,
                  AC_PROCEDURAL_ERROR,
                  AC_UNSPECIFIED_REASON,
                  AC_CABLE_TAMPER,
                  AC_INTRUSION_DETECTION,
                  AC_AUTHENTICATION_FAILURE,
                  AC_BREACH_OF_CONFIDENTIALITY,
                  AC_NON_REPUDIATION_FAILURE,
                  AC_UNAUTHORIZED_ACCESS_ATTEMPT,
                  AC_DELAYED_INFO,
                  AC_KEY_EXPIRED,
                  AC_OUT_OF_HOURS_ACTIVITY
            };

            /* This is a preliminary definition of an alarm record for an
audit log. */

            struct RecordType {
                  EventType event;
                  CauseType cause;
                  SeverityType severity;
                  string serviceUser;
                  string serviceProvider;
                  string additionalInfo;
                  string alarmDetector;
            };

            /* This interface is implemented by the user of a the security
service to receive alarm indications. */

            interface User {
                  /* This operation signals the user that a crypto alarm has
occurred.
                  @roseuid 39E47E7500CD */
                  void signalAlarm (
                        in string qualifier
                        );

            };

      };

      module Crypto {

            /* Identifies how a channel is configured. */

            enum ChannelType {
                  CT_SIMPLEX_RX,                /* Receive only operation. */
```

```
                  CT_HALF_DUPLEX,               /* The channel supports both
transmit and receive but only one at a time (the crypto will context switch
between receive and transmit portions of algorithm. */
                  CT_FULL_DUPLEX,               /* The channel is configured
for simultaneous receive and transmit (e.g. not context switching). */
                  CT_BLACK_SIDE,                /* This configures a channel
for black-black encrypt and decrypt (e.g. DAMA orderwire, loading of
classified waveforms). */
                  CT_RED_SIDE             /* This configures a channel for
red-red encrypt and decrypt. */
            };

            /* This structure is used to configure the crypto for operation
and to indicate the configuration of an instantiated channel. */

            struct ChannelConfigType {
                  /* The channel type.  Can translate into multiple channels
internally to the crypto device. (e.g. full duplex(. */
                  ChannelType type;
                  /* The ID of the crypto algorithm to use for the channel.
*/
                  IdType algorithm;
                  /* The key(s) to use for the channel.  Certain waveforms
require the use of multiple keys. */
                  IdSequenceType keys;
                  /* Only valid for CT_BUSS_FILL */
                  IdType certificate;
                  /* The set of modes in which the algorithm will operate. */
                  CF::StringSequence modes;
                  /* The set of properties for the algorithm such as straps,
seed, etc. */
                  CF::Properties properties;
                  IdType bypassPolicy;
            };

            interface SingleChannel {
                  /*
                  @roseuid 3A04198A01B8 */
                  oneway void transform (
                        in any bypass,
                        in CF::OctetSequence payload
                        );

            };

            interface MultiChannel {
                  /*
                  @roseuid 3A09586D02B7 */
                  oneway void transform (
                        in ChannelIdType channel,
                        in any bypass,
                        in CF::OctetSequence payload
                        );

            };
```

```
            exception ChannelInAlarm {
            };

            exception DeviceError {
            };

            exception InvalidChannelId {
            };

            exception UnknownError {
            };

            /* This interface supports black-black and red-red encryption and
decryption only.  Baseband data uses an instantiation of the
            packet interface for red-black encryption and black-red
decryption. */

            interface MultiChannelSingleSided {
                  /* Encrypt data using instantiated channel and return in
the same octet sequence.
                  @roseuid 39E7175B00F4 */
                  void Encrypt (
                        in ChannelIdType channel,
                        inout CF::OctetSequence data
                        )
                        raises (ChannelInAlarm, DeviceError,
InvalidChannelId, UnknownError);

                  /* Decrypt data using instantiated channel and return in
the same octet sequence.
                  @roseuid 39E71796030C */
                  void Decrypt (
                        in ChannelIdType channel,
                        inout CF::OctetSequence data
                        )
                        raises (ChannelInAlarm, DeviceError,
InvalidChannelId, UnknownError);

            };

            /* This interface supports black-black and red-red encryption and
decryption only.  Baseband data uses an instantiation of the
            packet interface for red-black encryption and black-red
decryption. */

            interface SingleChannelSingleSided {
                  /* Encrypt data using instantiated channel and return in
the same octet sequence.
                  @roseuid 3A082E5403B8 */
                  void Encrypt (
                        inout CF::OctetSequence data
                        )
                        raises (ChannelInAlarm, DeviceError, UnknownError);

                  /* Decrypt data using instantiated channel and return in
the same octet sequence.
```

```
                @roseuid 3A082E5403CD */
                void Decrypt (
                      inout CF::OctetSequence data
                      )
                      raises (ChannelInAlarm, DeviceError, UnknownError);

          };

          /* This interface supports crypto channel creation and
destruction. */

          interface Controller {
                exception AssuranceLevel {
                };

                exception InvalidKeyId {
                };

                exception InvalidAlgorithmId {
                };

                exception InvalidMode {
                };

                exception InvalidProperty {
                };

                exception ChannelAlreadyStarted {
                };

                exception ChannelNotStarted {
                };

                exception InvalidCertificateId {
                };

                exception CertificateNotRequired {
                };

                exception ChanTypeAlgorithmMismatch {
                };

                exception InvalidPolicyId {
                };

                exception NotCOMSECAlgorithm {
                };

                exception ResourcesUnavailable {
                };

                exception KeyAlgorithmMismatch {
                };

                exception KeyExpired {
                };
```

```
                  /* Creates a crypto channel and returns a channel ID.
                  @roseuid 39DA02E70354 */
                  ChannelIdType createChannel (
                        in ChannelConfigType configInfo
                        )
                        raises (AssuranceLevel, CertificateNotRequired,
ChanTypeAlgorithmMismatch, DeviceError, InvalidAlgorithmId,
InvalidCertificateId, InvalidKeyId, InvalidMode, InvalidPolicyId,
InvalidProperty, KeyAlgorithmMismatch, NotCOMSECAlgorithm,
ResourcesUnavailable, UnknownError);

                  /* Destroys an instantiated crypto channel.
                  @roseuid 39DA030500E0 */
                  void destroyChannel (
                        in ChannelIdType channel
                        )
                        raises (InvalidChannelId,UnknownError);

                  /* Gets the configuration of an instantiated crypto
channel.
                  @roseuid 39E36EF80052 */
                  void getChannelConfig (
                        in ChannelIdType channel,
                        out ChannelConfigType configInfo
                        )
                        raises (InvalidChannelId);

                  /*
                  @roseuid 3A045106027B */
                  void startChannel (
                        in ChannelIdType channel,
                        in string mode
                        )
                        raises
(ChannelAlreadyStarted,DeviceError,InvalidChannelId,InvalidMode,UnknownError)
;

                  /*
                  @roseuid 3A04511F0262 */
                  void stopChannel (
                        in ChannelIdType channel,
                        in string mode
                        )
                        raises
(ChannelNotStarted,DeviceError,InvalidChannelId,InvalidMode,UnknownError);

                  /*
                  @roseuid 3A0451260032 */
                  void resetChannel (
                        in ChannelIdType channel
                        )
                        raises (DeviceError,InvalidChannelId,UnknownError);

                  /*
                  @roseuid 3A04524E0128 */
```

```
        void resetCrypto ();

    };

};

module Certificate {

    /* This is the interface for management of certificate fills. */

    interface Manager : Fill::Manager {
    };

};

module Policy {

    interface Manager : Fill::Manager {
        /*
        @roseuid 39EF73350292 */
        CORBA::Policy getPolicy (
            in IdType id
            )
            raises (InvalidId);

    };

    interface AccessControlPolicy : CORBA::Policy {
    };

};

module Time {

    interface Manager {
        exception InvalidValue {
        };

        /*
        @roseuid 3A04573F02FD */
        void setTime (
            in TODType time
            )
            raises (InvalidValue);

        /*
        @roseuid 3A045747036D */
        TODType getTime ();

        /*
        @roseuid 3A04575200C0 */
        void setDate (
            in DateType date
            )
            raises (InvalidValue);
```

```
                /*
                @roseuid 3A04575702C6 */
                DateType getDate ();

            };

        };

        module GPS {

            interface Manager : Transec::Manager {
            };

        };

};

#endif
```